UNIVERSIDAD NACIONAL DE INGENIERIA

FACULTAD DE INGENIERIA ELECTRICA Y ELECTRONICA

TESIS:

"DISEÑO DE UNA RED DE DATOS FLEXIBLE UTILIZANDO SDN PARA REDUCIR EL CONSUMO DE ENERGÍA EN UN CENTRO DE DATOS"

PARA OBTENER EL GRADO ACADÉMICO DE MAESTRO EN CIENCIAS CON MENCIÓN EN TELEMÁTICA

ELABORADO POR:

RENATTO GUSTAVO GONZALES FIGUEROA

ASESOR:

Dr. Ing. JORGE BUTLER BLACKER

LIMA – PERÚ

2021

DEDICATORIA

Dedico esta tesis a mis hijos y esposa quienes me han dado su apoyo incondicional y su paciencia durante la elaboración de este documento.

AGRADECIMIENTOS

Un agradecimiento especial al asesor de mi tesis Dr. Ing. Jorge Butler Blacker, por sus consejos y apoyo brindado los cuales me han sido muy útiles para llevar a cabo el desarrollo de esta tesis. A mi gran amigo Marco Holguín quién contribuyo con sus conocimientos sobre Centros de Datos, parte clave en el desarrollo de esta tesis. A mi amigo y mentor Alfredo Rodríguez por su tiempo y consejos que contribuyeron a darle claridad a este trabajo.

INDICE DE CONTENIDOS

INDICE DE TABLAS

INDICE DE FIGURAS

RESUMEN

La eficiencia energética de un centro de datos está en razón de la energía consumida total por el centro de datos y la energía consumida por los sistemas de cómputo, conocido como PUE. En el presente proyecto se desarrolla una alternativa para reducir el consumo de energía en un centro de datos, manteniendo encendidos solo los recursos de red que se utilizan para cursar el tráfico que atraviesa la red en un momento dado. Para esto, nos enfocamos específicamente en los dispositivos de conectividad: conmutadores, sus puertos y los enlaces que los conectan; y no el resto de la infraestructura.

Actualmente las tecnologías de red están adoptando un nuevo paradigma: las redes definidas por software. Con ellas se abre la posibilidad de controlar, de manera centralizada, dispositivos de red a través de aplicaciones que pueden ser desarrolladas por los administradores de red, automatizar tareas y personalizar el comportamiento de la red. En este proyecto aprovecharemos la flexibilidad y programabilidad que nos ofrecen las SDN para gestionar el encendido y apagado de algunos recursos de la red del centro de datos con el fin de disminuir el consumo de energía de este.

El uso de un algoritmo de búsqueda de rutas es necesario para identificar el camino posible que se le puede asignar a un flujo de tráfico y garantizar que los recursos de red en ese camino estén disponibles. El algoritmo A* es el seleccionado para proyecto, es un algoritmo de búsqueda informada, heurístico que no solo utiliza el coto como una función heurística del camino por recorrer sino también considera el costo del camino ya recorrido.

Por lo tanto, se ha desarrollado una solución ElasticTree basada en SDN, programando una aplicación en Python que utiliza el algoritmo A* para seleccionar una ruta para el tráfico entrante nuevo. Una vez seleccionada la ruta el controlador actualiza, a través del protocolo OpenFlow, las tablas de flujo de los conmutadores que formaran parte de la ruta seleccionada. Utilizando una herramienta de cálculo como OCTAVE, aplicamos un modelo energético para calcular la variación del consumo de energía al mantener solamente encendidos los equipos necesarios, consiguiendo ahorros hasta de un 30% frente a una infraestructura rígida tradicional.

ABSTRACT

The energy efficiency of a data center is a ratio of the total energy consumed by the data center and the energy consumed by the computer systems, known as PUE. In this project, an alternative is developed to reduce energy consumption in a data center, keeping on only the network resources that are used to carry the traffic that crosses the network at any given time. To accomplish this task, we focus specifically on connectivity devices: like switches, their ports and the links that connect them; and not the rest of the infrastructure.

Today network technologies are embracing a new paradigm: software-defined networks. With them, the possibility of centrally controlling network devices is opened through applications that can be developed by network administrators, automate tasks, and customize network behavior. In this project, we will take advantage of the flexibility and programmability that SDNs offer us to manage and keep on or off network resources of the data center, in order to reduce its energy consumption.

The use of a route search algorithm is necessary to identify the possible route that can be assigned to a traffic flow and to ensure that the network resources on that route are available. A * is the algorithm selected for the project, it is an informed heuristic search algorithm that not only uses the limit as a heuristic function of the way to go but also considers the cost of the way already traveled.

Therefore, an ElasticTree solution based on SDN has been developed, programming a Python application that uses the A * algorithm to select a route for new incoming traffic. Once the route is selected, the controller updates, through the OpenFlow protocol, the flow tables of the switches that will be part of the selected route. Using a calculation tool such as OCTAVE, we apply an energy model to calculate the variation in energy consumption, keeping only the necessary equipment on, achieving savings of up to 30% compared to a traditional rigid infrastructure.

INTRODUCCIÓN

Los centros de datos son el centro neurálgico de las organizaciones, procesan una gran cantidad de datos y la distribuyen a los usuarios locales y remotos. Los centros de datos son la infraestructura que esta debajo de las aplicaciones que consumimos actualmente incluyendo la nube. Normalmente imaginamos un centro de datos como un conjunto de servidores alojados en racks procesando información en grandes cantidades, sin embargo, un centro de datos es más que solo servidores; otro componente clave es la red. La red de un centro de datos interconecta los servidores con otros servidores y con el exterior donde están los usuarios.

De acuerdo con el estudio realizado por Yang T., Lee Y., Zomaya A. - 2014 en [29], sobre la eficiencia energética en centros de datos. Podemos afirmar que más de un tercio de la energía es consumida por los enlaces de comunicaciones, los conmutadores y otros elementos de la red. Las nuevas iniciativas para reducir el consumo de energía de un centro de datos se están enfocando en esta área, controlar el tráfico que pasa por esta red se vuelve una necesidad.

Por otro lado, en el entorno de las redes de computadoras hay un paradigma que apareció hace más de diez años, que está tomando cada vez más relevancia y los fabricantes de equipos de comunicaciones ya tienen un abanico de productos que se pueden implementar. Nos referimos a las Redes Definidas por Software o SDN por sus siglas en inglés, este nuevo paradigma, ofrece la separación del plano de datos y el plano de control. De esta manera podemos mantener la infraestructura, plano de datos, como esta y trabajar en un plano de control centralizado, altamente configurable y automatizado. Las SDN nos dan la oportunidad de programar la red, así es posible atender de forma dinámica eventos y responder a ellos.

La facilidad de programación también permite integrar otras aplicaciones y código como algoritmos u otros. Para conseguir esto, la arquitectura de la SDN soporta comunicación a través de dos puentes: el puente norte, una interface para comunicarse con aplicaciones desarrolladas por terceros, normalmente utiliza APIs, como Restful API. El puente sur, una interfase para comunicarse con plano de datos, originalmente utiliza OpenFlow u otros protocolos como BGP, netflow, netconf, entre otros.

Actualmente las SDN están soportadas por organizaciones como la ONF (Open Network Foundation) que mantiene el protocolo OpenFlow e incluso ha desarrollado un controlador de código abierto llamado ONOS; y también por fabricantes como CISCO, Juniper, HP, Pica8. También existe una variedad de controladores disponibles como ONOS, ODL, Ryu, POX, Floodlight que van desde herramientas orientadas a la investigación y pruebas, como la operación en entornos comerciales de producción a nivel de Carrier.

Una alternativa de solución al problema energético en un centro de datos puede estar en la gestión de su red. Debido a la importancia que tiene el centro de datos, una de las características principales es la alta disponibilidad; lo que también se refleja en su red; esto se traduce en equipos y enlaces redundantes que permanecen encendidos esperando cursar tráfico en caso de que falle un enlace. Como se puede observar, la red de un centro de datos se configura con enlaces redundantes que permanecen activos consumiendo energía, a pesar de que no están siendo utilizados.

En este proyecto se plantea aprovechar las ventajas de las redes definidas por software, la programabilidad y la flexibilidad, para gestionar el tráfico que cursa la red de un centro de datos mediano y poder identificar que enlaces, puertos y conmutadores no necesitan estar encendidos en un momento especifico. La red SDN en consecuencia nos debería permitir encender y apagar recursos bajo demanda. En consecuencia, esto nos podría llevar a un ahorro de energía importante en función de los periodos de bajo uso del centro de datos.

CAPITULO I
ANTECEDENTES Y DESCRIPCION DEL PROBLEMA

En este capítulo se presentan los aspectos introductorios del proyecto se analiza la problemática energética en un centro de datos y se hace una introducción al nuevo paradigma en las redes de datos que son las SDN, Redes Definidas por Software por sus siglas en inglés. Con estos dos elementos establecemos el objetivo general y los objetivos específicos que nos permitirán completar con éxito este proyecto.

1.1. Antecedentes bibliográficos

Los Centros de Datos al concentrar el poder de cómputo, almacenamiento y comunicaciones para el despliegue de los servicios que utilizan las organizaciones, generan un alto consumo de energía. Según Schneider Electric en [27]: “Un centro de datos de alta disponibilidad de 1 MW puede consumir US$20.000.000 en electricidad durante su vida útil”, por lo que en muchos casos el costo de electricidad resulta ser mayor que el costo del hardware. Según el Departamento de energía de los Estados Unidos, el consumo de un centro de datos no es nada despreciable, puede requerir de 100 a 200 veces por metro cuadrado, la energía que consume una oficina común.

Por este motivo se han realizado diversas mejoras en los equipos para ser más eficientes energéticamente. Normalmente estas mejoras se han dado en los servidores, considerando el auge de la virtualización, es posible consolidar varios servidores virtuales en un solo hardware que optimiza el uso de recursos de cómputo, lo que trae como consecuencia un ahorro considerable de espacio y sobre todo energía.

Según Schneider Electric otro punto clave para mejorar el rendimiento energético de un centro de datos es la gestión de la temperatura, es decir el sistema de enfriamiento del centro de datos. Este es un punto crucial en el diseño de centros de datos ya que el manejo adecuado de la temperatura también permite un ahorro considerable de energía.

Para la presente tesis nos enfocaremos en el consumo de energía que generan los equipos de conectividad, es decir los conmutadores, enrutadores, puntos de acceso y otros dispositivos intermedios que formen parte de la red de un centro de datos. Aunque en la mayoría de los estudios sobre el consumo de energía no se toca esta parte, el consumo energético de la red es considerable como para tomarlo en cuenta.

Ashraf, F., Aslam, M., & Khan, Y. D. en [2] estudian diferentes marcos de trabajo relacionados con el ahorro de energía utilizando SDN y mencionan: “Las ventajas del paradigma SDN son reducción de energía, alta productividad, simplicidad de uso y sobre todo la rentabilidad” (p76).

Por lo mencionado en el párrafo anterior, se propone utilizar una plataforma flexible que centraliza el control de las funcionalidades de red en plano (Plano de Control), separando el flujo de datos basado en el reenvío de paquetes (Plano de Datos) de cada dispositivo de red. Esta separación permite programar, con mayor facilidad, el comportamiento de la red a través del envío de instrucciones a los equipos de conectividad. La programación se lleva a cabo a través de software, consiguiendo flexibilidad, robustez y facilidad de escalamiento ante la necesidad de cambios rápidos en la red. Esta nueva tendencia se le conoce como Redes Definidas por Software o SDN (Software Defined Networks) por sus siglas en ingles.

Los beneficios de utilizar SDN se hacen evidentes, ya no es necesario trabajar únicamente con protocolos establecidos y en muchos casos antiguos. Ya no es necesario esperar a que los fabricantes desarrollen características especializadas que encajen con nuestras necesidades. SDN nos permite implementar las características que necesitamos y a la medida de las necesidades de nuestra red.

Por lo antes mencionado la SDN se convierte en el aliado perfecto para conseguir el objetivo de este proyecto. Su capacidad de modificar el comportamiento de la red de manera flexible a través de programas de software desarrollados en diversos lenguajes de programación, nos permiten controlar los parámetros de un dispositivo de conectividad a nuestra voluntad; reaccionando ante cambios a través del tiempo. De esta manera podríamos apagar y encender: puertos, tarjetas de línea e incluso dispositivos completos a medida que el flujo de información atraviesa la red; proponiendo el ahorro en consumo de energía por dispositivo.

1.2. Descripción de la realidad problemática

Los centros de datos son la infraestructura que soporta las operaciones de las organizaciones actuales. Son el centro del procesamiento, almacenamiento y comunicaciones de la información que organizaciones y personas consumimos a diario. Los centros de datos alojan una gran cantidad de equipamiento: servidores, conmutadores, ruteadores, equipos de enfriamiento, entre otros; que consumen energía en grandes cantidades. Los centros de datos de los lideres tecnológicos como Google, Amazon, Facebook, Microsoft entre otros, suelen estar diseñados con mucha eficiencia; son los centros de datos de las organizaciones pequeñas los que suelen presentar problemas de consumo de energía.

A pesar de que las empresas son conscientes de que deben disminuir el consumo de energía en sus servidores, la necesidad de confiabilidad y disponibilidad es tan grande, que se suele preferir el exceso a la ausencia de recursos. Como consecuencia de esto, los esfuerzos por fabricar servidores eficientes y con bajo consumo energético son cada vez más exitosos; también encontramos mejores soluciones de climatización que buscan mantener el consumo de energía bajo.

Sin embargo, las soluciones antes mencionadas no se enfocan en otro elemento importante que forma parte de los centros de datos; la red. Cuantos más equipos estén conectados y cuantos más enlaces redundantes se puedan permitir se garantiza la disponibilidad de los servicios, pero se aumenta el consumo de energía en función de equipos conectados y puertos en funcionamiento. Sin embargo, estos enlaces no son utilizados todo el tiempo, desperdiciándose energía en puertos y enlaces que no conducen flujos de datos en un momento dado.

1.3. Formulación del problema

En la presente tesis se busca determinar en qué medida es posible ahorrar energía en la red de un centro de datos, orientado al aprovisionamiento de SaaS, controlando bajo demanda el número de puertos activos en los equipos de conectividad en función a los flujos de tráfico que intercambian los equipos de red, utilizando redes definidas por software.

1.4. Justificación e importancia de la investigación

La presente investigación se justifica desde el punto de vista científico ya que se propone un método para la gestión de puertos de un dispositivo de conectividad, en función al flujo de datos que atraviesa por cada enlace, empleando un algoritmo de búsqueda informada heurístico como A*.

Desde el punto de vista ecológico propone reducir el consumo energético en un centro de datos en un área en la que no hay soluciones comerciales eficientes actualmente que ataquen el problema. La mayoría de las soluciones apuntan a reducir el consumo de energía con un eficiente manejo del clima del centro de datos y con servidores eficientes; no toman en cuenta los equipos de conectividad.

Desde el punto de vista tecnológico se hará uso de un nuevo paradigma en las redes de datos, las Redes Definidas por Software, que al separar el plano de control del plano de datos de la red nos abren muchas posibilidades para controlar y automatizar las tareas de gestión y operación de la red.

1.5. Objetivos

1.5.1. Objetivo General

Diseñar una red de datos flexible utilizando Redes Definidas por Software que gestione la habilitación de los puertos de comunicaciones de los equipos de conmutación en función del flujo de tráfico para reducir el consumo de energía de un Centro de Datos.

1.5.2. Objetivos Específicos

- Determinar la topología utilizada para simular la red del centro de datos e identificar los puertos que serán administrados por la SDN.
- Determinar el algoritmo de optimización para establecer los puertos que deberán utilizarse durante la operación de la red del centro de datos.
- Determinar la fórmula que se utilizará para calcular el consumo de energía de conmutadores con sus puertos encendidos y/o apagados.
- Establecer la arquitectura que se empleara para la simulación: controladores y protocolos de comunicación.

- Simular la red SDN en un entorno virtual introduciendo tráfico de prueba.
- Validación de los resultados de la simulación calculando el consumo de energía de la red del centro de datos en función de los puertos encendidos y/o apagados.

1.6. Hipótesis

1.6.1. Hipótesis General

Si aplicamos el paradigma de redes definidas por software para crear una red flexible podríamos mejorar las posibilidades de reducir el consumo de energía de los dispositivos de conectividad en la red de un Centro de Datos.

1.6.2. Hipótesis Especificas

Es posible establecer los flujos de tráfico bajo demanda entre los conmutadores de la red, para controlar que puertos de comunicación estarán en uso o no durante un determinado tiempo.

Mantener los puertos de comunicación de un conmutador apagados cuando no están en uso contribuye a la reducción del consumo de energía en la red de un centro de datos.

1.7. Variables e Indicadores

1.7.1. Variable Independiente

Flexibilidad de la red para gestionar los flujos de tráfico bajo demanda, utilizando redes definidas por software.

Indicador:

Actualización de la tabla de flujos de los conmutadores a medida que aparecen nuevos flujos de tráfico en la red.

1.7.2. Variable Dependiente

Consumo de energía en la red de un centro de datos.

Indicador:

Reducción del consumo de energía en comparación con una topología fija como FatTree.

1.8. Unidad de análisis

Se simulará la red de un centro de datos con una topología FatTree con n = 4, con mininet. El escenario de análisis será: 16 servidores, 21 conmutadores y 84 puertos. Se tomarán los datos y se analizarán con OCATVE para calcular el ahorro de energía.

1.9. Tipo y nivel de investigación

La investigación es de tipo aplicada y experimental, ya que a través de la simulación utilizando como herramienta mininet (simulador de redes que puede conectarse a un controlador SDN) se creará una red de datos virtual con las características adecuadas para un Centro de Datos. Sobre esta red virtual se probarán los scripts desarrollados en Python para identificar flujos de datos, puertos activos/inactivos y gestionar en función de esta información el apagado o encendido bajo demanda de los puertos de comunicación de los dispositivos de conectividad, lo que trae como consecuencia el ahorro de energía.

Se utilizará una herramienta de programación matemática como OCTAVE donde ejecutaremos las fórmulas de consumo de energía en conmutadores para evaluar de manera cuantitativa el ahorro de energía de acuerdo con las decisiones que tome el controlador SDN. Se realizarán proyecciones en el tiempo y se evaluara el ahorro energético mensual.

1.10. Periodo de análisis

Se utilizará un script automatizado, que simulará las solicitudes de tráfico en una red por un periodo de una hora. Esto nos permitirá simular escenarios en horas pico u horas de menor tráfico.

1.11. Fuentes de información e instrumentos utilizados

Las fuentes de información son artículos científicos relacionados a la gestión de energía en centros de datos y aplicación de las SDN en redes de datos flexibles; también se utilizarán artículos, hojas de datos y otros documentos de fabricantes de equipos de red como CISCO y libros de redes definidas por software.

Se utilizaron las herramientas Schoolar® Google, Scopus y Mendeley para la búsqueda de artículos como fuente de información.

1.12. Técnicas de recolección y procesamiento de datos

- Creación de matrices que almacenaran la información de rutas, tráfico, conexiones y direcciones en memoria en tiempo de ejecución.
- Archivos en formato csv donde se exportan los cambios de topología para ser analizados.
- El procesamiento se realizará utilizando una herramienta matemática como OCTAVE, utilizando la matriz de cambios de topología en formato csv, OCTAVE calculará el ahorro de energía.

CAPITULO II
MARCO TEÓRICO Y MARCO CONCEPTUAL

Para poder presentar el proyecto es necesario hacer una breve revisión a varios conceptos y tecnologías que serán empleadas durante su desarrollo.

En este capítulo se desarrollan conceptos sobre la arquitectura de las Redes Definidas por Software y las herramientas necesarias para la simulación de sus componentes. Se estudian los modelos topológicos utilizados para implementar redes en los centros de datos, así como los modelos matemáticos para el cálculo de energía en su red de conmutadores. También revisaremos conceptos de algoritmos de optimización de búsquedas informadas de rutas que utilizaremos para establecer el mejor conjunto que satisfaga las condiciones de tráfico y de consumo de energía.

2.1. Redes Definidas por Software (SDN)

Martin Casado, el padre de las Redes Definidas por Software en una entrevista a InternetNews en 2013 en [13] dice: "…ya no sé qué es SDN, cuando la cree era algo muy específico, pero [actualmente] se ha convertido en un término general que incluye múltiples visiones de lo que SDN es, de las cuales, no todas incluyen OpenFlow…".

CISCO en [4] propone nuevos servicios basados en SDN con la siguiente propuesta: "Automatiza y Programa tu red rápido. Provisiona, gestiona y programa redes rápidamente con SDN. (…) SDN provee opciones de automatización y programabilidad a través de centros de datos, redes de campus y redes de área amplia. Utiliza soluciones definidas por software de cisco, implementa redes basadas en intención".

Para la ONF (Open Network Foundation) en [18] la SDN es: "Es la separación física del plano de control y el plano de reenvío de datos en una red, donde el plano de control se encarga de controlar un gran número de dispositivos."

Las Redes Definidas por Software o SDN (en inglés: Software Defined Networking) son la evolución de la gestión de recursos de red. En sus inicios los mensajes de control y

datos de la red eran enviados por el mismo canal, normalmente a través de las redes telefónicas utilizando frecuencias especiales para tomar y liberar líneas, por ejemplo. Esto hace las redes muy inseguras.

En 1981 se introduce el Punto de Control de Red (NCP Network Control Point) primera implementación de control centralizado ideado para soportar un amplio rango de aplicaciones de gestión de red. Trayendo como beneficio manejar servicios bajo demanda y la capacidad de introducir servicios de manera rápida. Elimina la señalización en banda disminuyendo los gastos de al disminuir los tiempos de establecimiento de llamada. Ahora es posible observar directamente el comportamiento de la red y la evolución de la infraestructura, los datos y los servicios es independiente.

El siguiente paso en 1990 son las redes activas, cuando se introducen elementos de conmutación que realizan procesos de cómputo sobre los paquetes que los cruzan, por ejemplo: ruteadores activos, proxies, firewalls, honeypots, entre otros. Sin embargo, este enfoque de red activa no prospero por que la tecnología para implementar los nodos activos era muy cara para la época, al enviar paquetes llevando código de comando era necesario mecanismos de seguridad que los lenguajes de programación de la época no contemplaban. Sin embargo, dejaron un legado para las redes definidas por software:

- Funciones de programación en la red que permiten la innovación.
- La posibilidad de demultiplexar programas en las cabeceras de los paquetes.
- Enfocarse en los middleboxes y cómo se desarrollan sus funciones.

Un paso más adelante, según la experiencia obtenida de las redes activas, fue separar el plano de control y el plano de datos. Como observamos en la figura 2.1 el concepto básicamente consiste en separar la red en un plano de control (la inteligencia) y un plano de datos (hardware enfocado en el reenvío de paquetes).

La separación de estos dos planos nos permite evolucionar independientemente el software para el control de la red y el hardware de reenvío y conmutación. Se puede por lo tanto tener mayor control utilizando software de alto nivel.

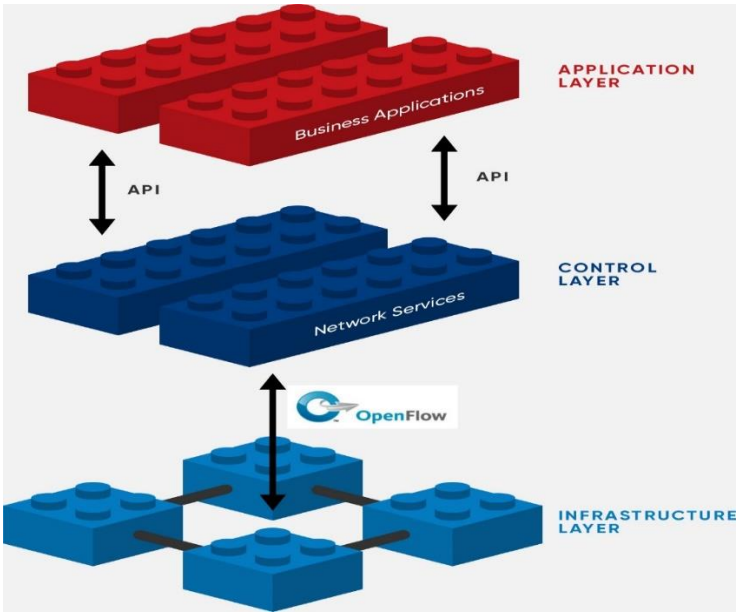


Figura 2.1: Arquitectura de la SDN según la ONF. (Fuente: ONF [18])

2.1.1. ONF – Open Network Foundation

Como se vio anteriormente las Redes Definidas por Software se han vuelto un término que abarca diferentes visiones del mismo enfoque, una red programable y automatizable a medida del cliente. Sin embargo, para entender mejor este enfoque, este proyecto se centrará en la visión de la ONF.

La ONF es una organización que promueve el desarrollo de protocolos y componentes de código abierto para las SDN, fue fundada por McKeown and Shenke en 2011 para que el desarrollo del protocolo OpenFlow esté en manos de una organización sin fines de lucro.

La Misión de la ONF tal como está en su sitio web en [18] dice:

> La ONF es un consorcio sin fines de lucro que conduce la transformación de la infraestructura de red y los modelos de negocios de las empresas de transporte de datos (Carriers). Es una comunidad de comunidades abierta y colaborativa. La ONF sirve como un paraguas para una serie de proyectos de desarrollo de soluciones aprovechando la desagregación de la red, la economía de cajas blancas (*white-box*), el software de código abierto y los estándares definidos por software para revolucionar la industria del transporte de datos (Carriers).

Desde el 2011 la ONF ha estado trabajando y ampliando el ecosistema para SDN a partir de OpenFlow, en la Figura 2.2 podemos observar su evolución.

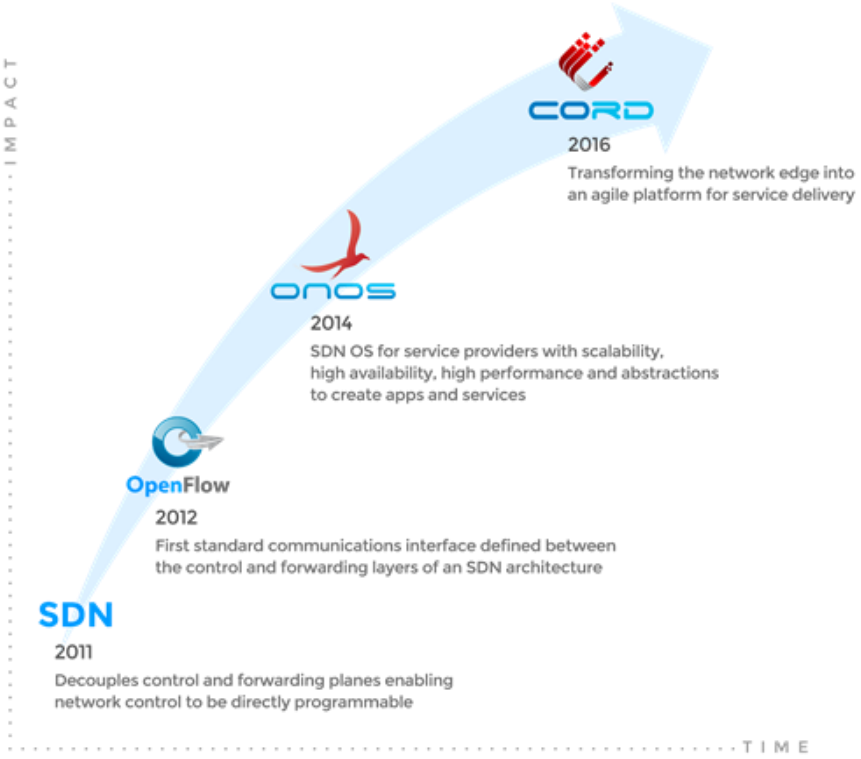


Figura 2.2: Evolución de la ONF.
(Fuente: ONF [18])

- 2011 SDN, separa el plano de control del plano de datos (plano de reenvío), permitiendo que el control de la red sea programable directamente.
- 2012 OpenFlow, el primer protocolo estándar de comunicación definido entre el plan de control y el plan de datos dentro de la arquitectura SDN.
- 2014 ONOS, Sistema Operativo SDN (controlador) para proveedores de servicios que provee escalabilidad, alta disponibilidad, alto desempeño y abstracciones para crear aplicaciones (apps) y servicios.
- 2016 CORD, una plataforma para operadores, que busca agilizar la entrega de servicios finales a sus clientes con propuestas innovadoras.

Las definiciones referidas a la SDN y su arquitectura serán tomadas de la ONF, así como los conceptos que forman parte del protocolo OpenFlow, que utilizaremos para implementar la SBI de este proyecto.

2.1.2. Definición

Como vimos en el punto anterior hay diferentes definiciones para SDN, Gilad J. en su blog en [7] nos propone dos visiones para definir una SDN:

> … la versión purista define una abstracción completa entre el plano de control y el plano de datos para permitir combinaciones “plug and play” de software y hardware independientes para alcanzar un máximo de innovación y velocidad…
>
> … la versión más pragmática se enfoca en alcanzar los mismos objetivos de automatización de una red basada en controlador y simplificando operaciones; al mismo tiempo que ofrece la solidez y el rendimiento requeridos por una organización… (2020)

La visión más purista está de acuerdo con la definición de la ONF que se vio anteriormente y será la que utilizaremos en este proyecto.

Una red definida por software es un nuevo enfoque para el diseño, implementación y gestión de redes. Para explicar el concepto debemos identificar dos planos que interactúan entre sí para hacer funcionar a la red de datos: El plano de datos, que se refiere al hardware encargado del reenvío de paquetes, esto se hace en función de la lógica de control. Y un plano de Control, donde se maneja la lógica que controlará el reenvío de paquetes.

Como observamos en la Figura 2.3 la arquitectura de una red tradicional concentra el plano de control y el plano de datos dentro de un mismo dispositivo. Es decir, cada equipo en la red cuenta con un hardware de reenvío de paquetes y un sistema operativo con aplicaciones que determinan las funciones que este dispositivo puede cumplir y por lo tanto que lógica utilizará para reenviar paquetes; esto incluye que protocolos y estándares puede soportar dicho dispositivo. Bajo este enfoque podemos observar, que cada dispositivo de red se debe configurar y gestionar de manera independiente para que reenvíe paquetes e interactúe con otros dispositivos de red. Por otro lado, en la arquitectura de una red SDN observamos que el plano de control y el plano de datos se desagregan verticalmente; centralizando el plano de control en una sola ubicación, el dispositivo controlador. Los

dispositivos de red solo se quedan con el plano de datos, es decir mantienen un sistema operativo, pero con la lógica mínima para reenviar paquetes y una aplicación (suele ser un agente) que se comunicara con el dispositivo controlador para recibir las instrucciones que utilizara para el reenvío de paquetes.

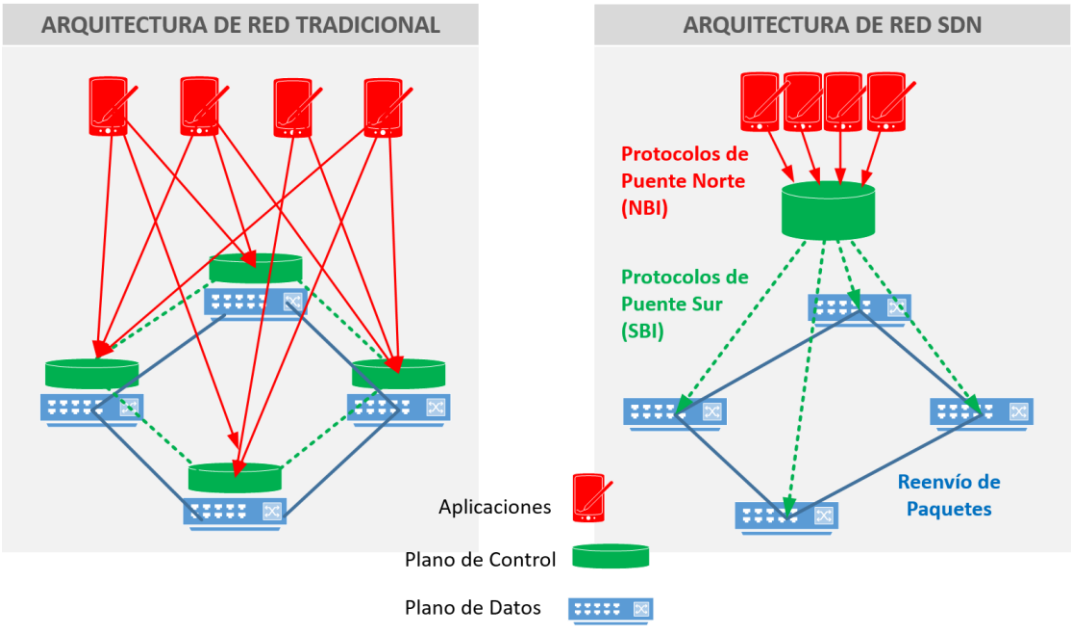


Figura 2.3: Arquitectura Tradicional frente a la Arquitectura SDN.

En este enfoque, el controlador provee una vista abstracta y centralizada de la red en conjunto. Es a través del controlador donde los administradores de red pueden, de manera rápida y fácil, desplegar decisiones de como los dispositivos de reenvío (ruteadores, conmutadores y otros) que forman parte del plano de datos deben manejaran el tráfico de la red. En la figura 2.3 podemos observar que en la arquitectura SDN, el controlador (en color verde) utiliza dos interfaces de comunicación: una orientada hacia los dispositivos del plano de datos y otra hacia las aplicaciones de red. La interface hacia las aplicaciones se le conoce como *NorthBound Interface* (NBI) y utiliza APIs para comunicarse con las aplicaciones que se encargan de facilitar la innovación y habilitar un servicio eficientemente orquestado y automatizado. La interface hacia el plano de datos se le conoce como *SouthBound Interface* (SBI o también conocido como CDPI: *Control-Data Plane Interface*) y utiliza protocolos que permitan llevar información entre los dispositivos del plano de datos y el controlador para mantener la gestión de la red. De esta manera una SDN nos facilita modelar el tráfico y desplegar servicios de tal manera que puedan mantenerse a la par de las necesidades cambiantes del negocio; sin tener que modificar cada conmutador o ruteador de manera independiente en el plano de datos.

Con esta arquitectura el diseño se vuelve más simple si la comparamos con una red tradicional, es posible independizarse de los fabricantes de dispositivos y simplificar la configuración de los equipos, ya que estos no necesitan ser configurados uno a uno, sino aceptar las instrucciones del dispositivo controlador.

Según el sitio web de OpenDaylight en [20], los beneficios de este enfoque son bastante obvios:

> … no va a ser necesario mantener el uso de protocolos antiguos. No más esperar y desear que los fabricantes desarrollen las características especializadas que necesitas. Con la posibilidad de desarrollar tus propias funcionalidades, ahora es posible optimizar la selección de dispositivos por precio y desempeño, independientemente de las funcionalidades especiales… (opendaylight.org, 2019)

En resumen, podríamos decir que el enfoque de SDN:

- Permite mantener un control y una gestión centralizada de los dispositivos de red de diferentes fabricantes.
- Al separar el plano de control permite que el plano de datos sea programable de una forma directa.
- Al proponer una red abierta que sea programable, se pueden crear servicios de forma sencilla.
- Las redes abiertas adoptaran mayor facilidad para las innovaciones.
- Mantiene la interoperabilidad entre diferentes fabricantes de dispositivos físicos o virtuales.
- La automatización personalizada de la red es una realidad.

2.1.3. Arquitectura de la SDN

Si bien es cierto, la SDN divide principalmente el plano de control y el plano de datos de manera funcional, podemos considerar que su arquitectura se conforma de 3 planos. La ONF nos presenta los 3 planos y su interacción tal como observamos en la figura 2.4:

- En lo más alto está el plano de aplicaciones, las aplicaciones son programas que se comunican con el controlador SDN a través del NBI solicitándole información acerca de la red y enviándole sus requerimientos sobre cómo debe comportarse la

red. Las aplicaciones son utilizadas directamente por los usuarios, procesan la información que reciben de la red para mostrarla de manera legible y reaccionar a los cambios que pueden presentarse en la topología o en el flujo de tráfico. También pueden enviar indicaciones al controlador para modificar los flujos de datos, seleccionando las mejores rutas, balanceando carga o filtrado paquetes en función del tráfico cursado.

- En lo más bajo esta el plano de datos conformado por los dispositivos de red, estos se encargan del reenvío de paquetes moviendo el tráfico de datos a través de la infraestructura. Estos dispositivos ya no necesitan tener un firmware específico para tomar decisiones de ruteo o conmutación; sino más bien interpretar las ordenes enviadas por el controlador para darle forma al tráfico de la red. Siguiendo las órdenes del controlador el plano de datos puede reenviar, descartar, replicar o procesar un paquete.
- En medio de estos dos planos está el plano de control, aquí encontramos al Controlador SDN como el elemento más importante de la red SDN. El controlador asume las funciones de control de manera centralizada utilizando software, lo que hace posible retirar las funciones de control que se ejecutaban de manera independiente en cada dispositivo. El controlador mantiene una vista de la red completa, tiene la capacidad de implementar políticas controlando a todos los dispositivos del plano de datos. Implementa APIs para el NBI de tal manera que pueda integrarse con las aplicaciones y se comunica utilizando protocolos (como OpenFlow), a través del SBI, con cada equipo del plano de datos (físico o virtual) facilitando la automatización de la gestión de la red y haciendo más fácil la integración y administración de las aplicaciones en las redes corporativas de las empresas.
- NBI (North Bound Interface), esta interface se establece como su nombre lo indica hacia el norte del controlador, es decir hacia las aplicaciones, por lo tanto, aquí se implementan APIs que sean compatibles con diferentes lenguajes de programación. Esto debe permitir que diferentes aplicaciones (de monitoreo, gestión, enrutamiento, etc) puedan comunicarse con el controlador y lo ideal es que sean implementadas de manera abierta e indistinta a los fabricantes.
- SBI (South Bound Interface, también conocido como CDPI), esta interface se establece hacia el sur de controlador, hacia el plano de datos, es decir los dispositivos que realizan el reenvío de paquetes. Se implementa a través de protocolos de comunicación que sean capaces de intercambiar información entre el

controlador y los agentes que residen en cada dispositivo. A través de esta interface se recolecta información de la red como estadísticas de tráfico o notificación de eventos y se envían comandos que actualizan las tablas de flujo, estados de dispositivos, entre otros.

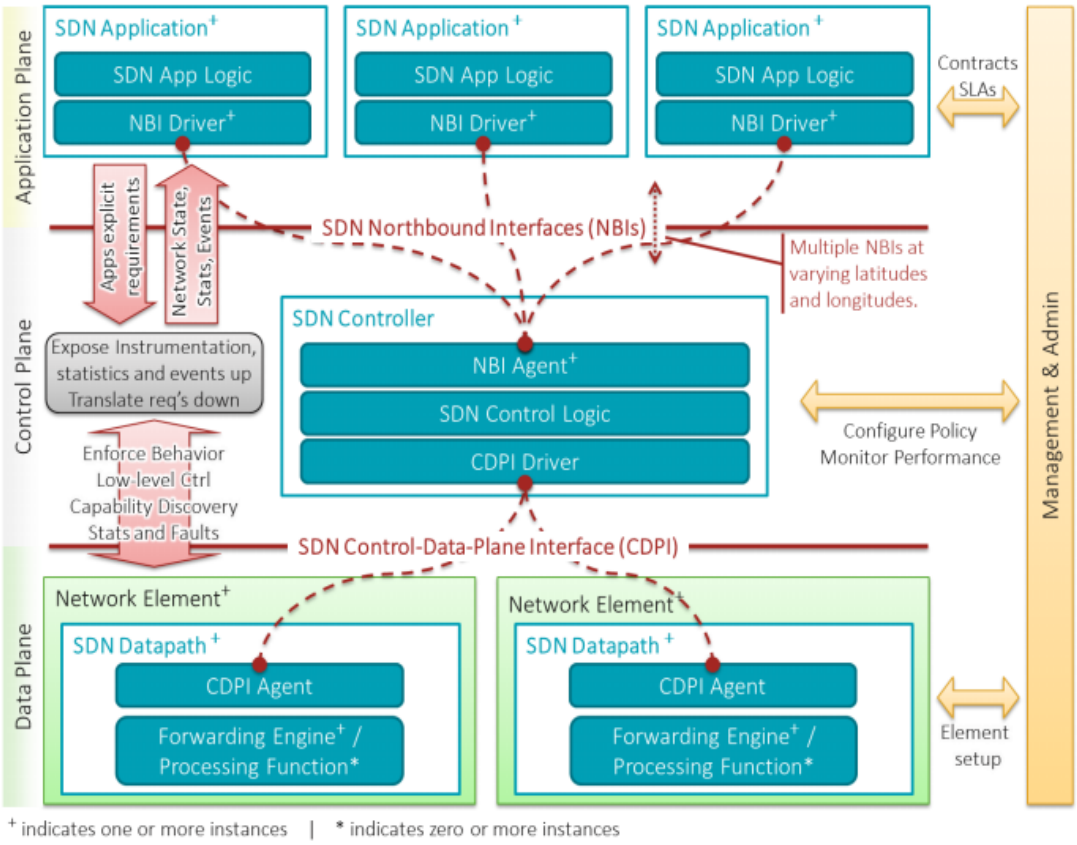


Figura 2.4: Arquitectura SDN: plano de datos, control y aplicación. (Fuente: ONF [17])

2.1.4. Ventajas de la Arquitectura SDN

Como se ha podido observar la principal característica de la arquitectura SDN es ser programable y automatizable, esto nos trae múltiples ventajas si la comparamos con la arquitectura de red tradicional. Después de revisar documentación de diferentes fuentes, la mayoría coincide en lo mismo, aquí mencionamos las ventajas que nos presenta la ONF:

- Programable directamente. El control de la red puede ser programado de manera directa ya que está separado de las funciones de reenvío de paquetes.

- Agil, la separación del control del reenvío de paquetes les permite a los administradores de red ajustar dinámicamente el tráfico que atraviesa la red para que coincidan con las necesidades del negocio.
- Manejo centralizado, la inteligencia de la red se centraliza de manera lógica en controladores basados en software, estos controladores mantienen una vista global de la red. Esta vista es compartida a las aplicaciones que ven a la red como un todo, como si fuera un único gran conmutador lógico.
- Configuración programable, la SDN permite a los administradores de red configurar, gestionar, asegurar y optimizar los recursos de red de manera rápida mediante programas automatizados; que pueden ser desarrollados por ellos mismos, sin depender de software propietario.
- Basada en estándares abiertos y neutral a fabricantes, cuando se implementa a través de estándares abiertos, el diseño y la operación se simplifican ya que no es necesario gestionar múltiples, protocolos y dispositivos propietarios de cada fabricante.

2.2. Controladores SDN

Como ya se ha mencionado anteriormente SDN es la separación del plano de datos del plano de control, y esta separación trae muchas oportunidades para los administradores de red, entre ellas, principalmente la programabilidad de la red. La pieza clave para conseguir tal objetivo es el controlador SDN, que reside en el plano de control y le proporciona el software que permite interactuar a las aplicaciones con el hardware de reenvió de paquetes.

Para entender mejor su funcionamiento utilizaremos una analogía utilizada por Siamak Azodolmolky en su libro: Software Defined Networking with OpenFlow, donde menciona que un controlador es como el Sistema Operativo de la red.

Azodolmolky en [3] dice: "… el controlador (de manera similar a un sistema operativo) les proporciona una interface programable a los conmutadores (similar al hardware de una computadora). Utilizando esta interface programable las aplicaciones de red (…) pueden ser escritas para realizar tareas de gestión y control ofreciendo nuevas funcionalidades…" (p.39).

Por lo tanto, siguiendo la analogía, el controlador como un sistema operativo de red debe implementar interfaces de comunicación con ambos extremos: los dispositivos de reenvío a través del SBI utilizando protocolos como OpenFlow y las aplicaciones a través del NBI utilizando APIs que soporten diferentes lenguajes de programación. La integración de aplicaciones que interactúen con el controlador aumenta significativamente las posibilidades de configuración e integración con otras plataformas.

Típicamente los controladores tienen una colección de módulos que realizan diferentes tareas en la red. Estas tareas pueden ser: inventario de dispositivos, recolección de estadísticas, ejecución de algoritmos, despliegue de reglas de control, selección de rutas, análisis de flujos, entre otras tareas. El controlador está orientado a los desarrolladores por lo que desde este punto de vista lo que se busca de un controlador es:

- Una base para crear código limpio (no necesariamente el uso de GUI).
- Un conjunto de componentes preconstruidos para gestionar la red con mayor facilidad. Por ejemplo, interface de consultas, acceso a la topología, etc.
- Debe tener un ciclo rápido de despliegue, es decir debería tomar solo unos minutos compilar los cambios y poner en producción el controlador.
- Desarrollo activo del código base, es decir un conjunto de desarrolladores continuamente arreglando y reparando el código base del controlador para tener siempre mejores versiones.
- Una activa comunidad de soporte a la que se pueda hacer consultas constantemente y que responda con rapidez.
- Que posea suficiente documentación.

En la actualidad existen muchos controladores en el mercado desde controladores simples y de código abierto, hasta controladores más complejos y propietarios de diferentes fabricantes. A continuación, mencionaremos algunos de los controladores más relevantes.

2.2.1. NOX

NOX es una plataforma para desarrollar aplicaciones SDN basadas en C++. Fue fabricado por Nicira Networks y desarrollado a la par del protocolo OpenFlow, por lo que se le considera el controlador OpenFlow original.

Características principales:

- Incluye una API en C++ y un API para Python.
- Posee una interface I/O asíncrona rápida. De tal manera que durante un instante de tiempo el controlador puede manejar flujos de datos grandes.
- Incluye componentes de ejemplo: para descubrimiento de topología,
- Su diseño permite instalarlo en diferentes distribuciones de Linux como CentOS, Ubuntu, Debian o RedHat.

Fue donado por Nicira Networks en el 2008 a la comunidad y se convirtió en código abierto; a partir de ahí ha servido de base para proyectos de investigación en SDN durante sus inicios. Una característica de NOX es que no solamente es una plataforma de programación de alto nivel para el control en SDN sino también un marco de desarrollo para aplicaciones

Según Sridhar Rao en [28] menciona que ha sido dividido en varias líneas de desarrollo:

- NOX Classic, es la versión que está disponible bajo licencia GPL desde 2019.
- NOX, conocido como “New NOX”, esta versión solo contiene soporte para C++, y aunque tiene menos aplicaciones que la versión classic, es mucho más rápida.
- POX, es una versión de NOX con soporte para Python.

2.2.2. POX

POX es una plataforma de código abierto basada en Python para desarrollar aplicaciones SDN, es una bifurcación de NOX que permite el prototipado rápido y desarrollo de aplicaciones, por lo que comenzó a llevarle ventaja a NOX.

En el sitio web de NOXrepo, menciona que el objetivo principal de POX es: “crear un arquetipo de controlador SDN moderno”. Características principales:

- Utiliza Pythonic, como interfase para OpenFlow.
- Se ejecuta en diferentes plataformas: Windows, Mac OS, Linux, etc. Utiliza como entorno de ejecución a “PyPy”.

- Trabaja con la GUI de NOX y también utiliza las mismas herramientas de visualización.
- Tiene componentes de ejemplo como: selección de rutas, descubrimiento de topología entre otros; que son reutilizables.
- Mejora el rendimiento de los programas NOX escritos en Python, por lo que reemplaza al NOX Classic.

2.2.3. BEACON

Fue creado en 2010 por David Erickson en la Universidad de Stanford, es un controlador SDN modular, rápido, y multiplataforma; escrito en Java, capaz de que soportar tanto la operación basada en eventos como la operación basada en multihilos.

David Erickson en [6] nos presenta a BEACON como: " …un controlador de código abierto basado en Java. Utilizado ampliamente en la enseñanza, investigación y utilizado ocmo base para [crear] Floodlight"(p13)

A partir del documento anterior listamos las características de BEACON:

- Su estabilidad ha sido probada en centros de datos, así como en diversos proyectos de investigación sin presentar caídas durante largos periodos de trabajo.
- Puede ejecutarse en diversas plataformas ya que al estar desarrollado en java solo necesita la JVM.
- Es de código abierto, la versión actual de BEACON 1.0.4 se encuentra actualmente bajo la licencia BSD.
- El manejo de librerías es dinámico, se puede gestionar el uso de las librerías en tiempo de ejecución.
- Opcionalmente se puede utilizar una interfaz web sobre Jetty Enterprise como plataforma web y con un framework configurable y extensible.
- Frameworks: está construido sobre frameworks en Java maduros como Spring y Equinox (OSGi)

2.2.4. Floodlight

Floodlight es otro controlador SDN escrito en java, soporta tanto conmutadores físicos como virtuales. Es una bifurcación de BEACON que ha sido rediseñada sin el framework

OSGi, por lo que puede ejecutarse y modificarse sin necesidad de utilizarlo. Se distribuye bajo licencia Apache y tiene una comunidad bastante grande de desarrolladores.

Harkal V. y Deshmukh A. en [11] nos dicen los objetivos de diseño de Floodlight:

- Alta disponibilidad, es posible implementar un cluster de controladores.
- Confiabilidad, esta se basa en el modo “activo-en espera” que se puede configurar por cada nodo del controlador, el establecer conexiones seguras entre el controlador y los nodos de conmutación; y la capacidad de manejar controladores múltiples basados en flujo abierto
- Manejar dispositivos que soporten tanto OpenFlow, como no.
- Diseñado para garantizar alto desempeño al estar desarrollado desde sus inicios como multihilo.
- Tiene soporte para la plataforma OpenStack.

2.2.5. OpenDaylight (ODL)

OpenDaylight es un proyecto colaborativo de la Fundación Linux para llenar la necesidad de desarrollar un framework abierto para la programabilidad y el control a través de una solución SDN de código-abierto. La comunidad que desarrolla ODL lo ha desarrollado pensando en que sea un componente central para cualquier arquitectura SDN, la idea es que permita a los usuarios reducir la complejidad de operación, extender el tiempo de vida de la infraestructura de su red y habilitar nuevas capacidades y servicios de manera flexible y robusta.

ODL en [20], menciona:

> “OpenDaylight (ODL) es una plataforma abierta modular para personalizar y automatizar redes de cualquier tamaño y escala. El proyecto OpenDaylight surgió del movimiento SDN, con un claro enfoque en la capacidad de programación de la red. Fue diseñado desde el principio como base para soluciones comerciales que abordan una variedad de casos de uso en entornos de red existentes.”

Como vemos ODL es de uso abierto, cualquiera puede: utilizarlo implementarlo y aprovechar sus funcionalidades o desarrollar y contribuir al código fuente. Se ejecuta en su propia JVM, lo que significa que puede funcionar sobre cualquier sistema operativo que

soporte java. Lo que lo hace una plataforma modular para personalizar y automatizar redes de cualquier tamaño.

Es un controlador bastante utilizado, Opendaylight en [21] podemos encontrar el enlace donde se puede descargar la última versión del controlador, la descripción y características nuevas.

2.2.6. RYU

Ryu es otro controlador que apareció aproximadamente en el 2013, su nombre significa “fluir” en japones, está basado en Python y es bastante flexible sobre todo para fines de investigación.

Según su página su sitio web de documentación ryu.readthedocs.io en [27]: “Ryu es un marco de trabajo (framework) para redes definidas por software basado en componentes. Ryu provee APIs bien definidas, para que los desarrolladores puedan crear aplicaciones. Soporta varios protocolos para gestión de redes como: Openflow, Netconf, OF-config, entre otros. EN OpenFlow soporta las versiones 1.0, 1.3, 1.4 y 1.5. Su código está bajo licencia Apache 2.0. ryu-sdn.org”.

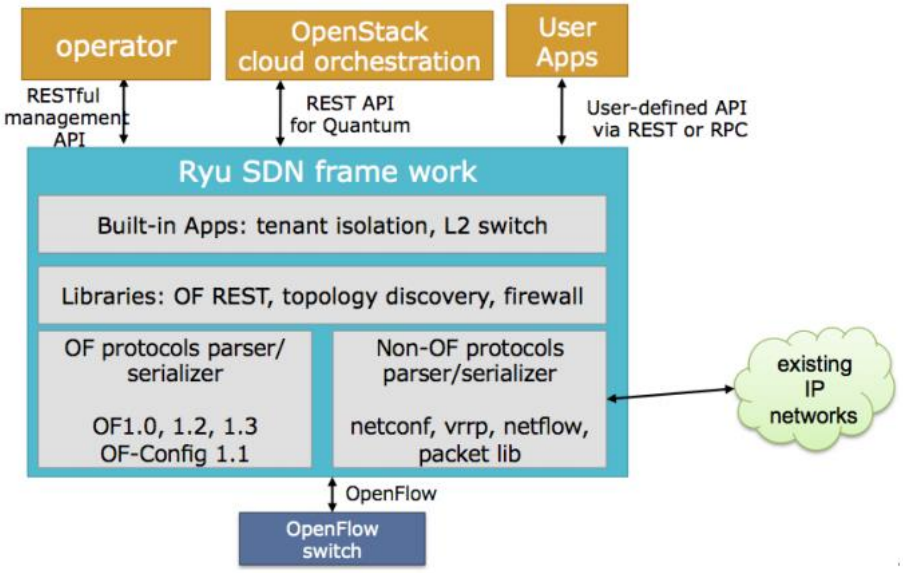


Figura 2.5: Arquitectura de RYU
(Fuente: Russell S., Pemberton D., Linton A. [25])

La arquitectura de RYU se puede ver en la figura 2.5, junto con todos sus componentes. Para comunicarse hacia el norte con las aplicaciones puede utilizar RESTful API, REST API, APIs definidas por el usuario vía REST o RPC. Hacia el sur se puede

comunicar con conmutadores OpenFlow, utilizando Openflow hasta la versión 1.5. Y para comunicarse con redes tradicionales puede utilizar netconf, netflow, vrrp.

Ryu se ejecuta en modo multihilo y utiliza *eventlets*; para realizar programas en RYU podemos utilizar entidades propias del controlador que se comunican entre sí de manera asíncrona a través de eventos.

Los Eventos, son objetos heredados de ryu.controller.event.EventBase, que se utilizan para enviar y recibir mensajes. Para poder recibir los eventos y luego atenderlos se utiliza una cola de eventos (event-queue), por cada aplicación se crea una sola cola para recibir eventos que está definida como FIFO.

Los eventos son manejados a través de un Manejador de eventos (Event Handler) que se ejecuta cuando ocurre algún evento especificado y llama al controlador para que ponga atención. La figura 2.6 nos muestra cómo funciona RYU, el proceso principal de ejecución es conocido como ryu-manager, podemos observar como el *event-handler* envía un evento a la cola (Event queue), donde el evento espera a ser ejecutado por el programa dentro de un hilo (Event loop Thread) que continuamente llama al manejador de eventos y así se repite el ciclo.

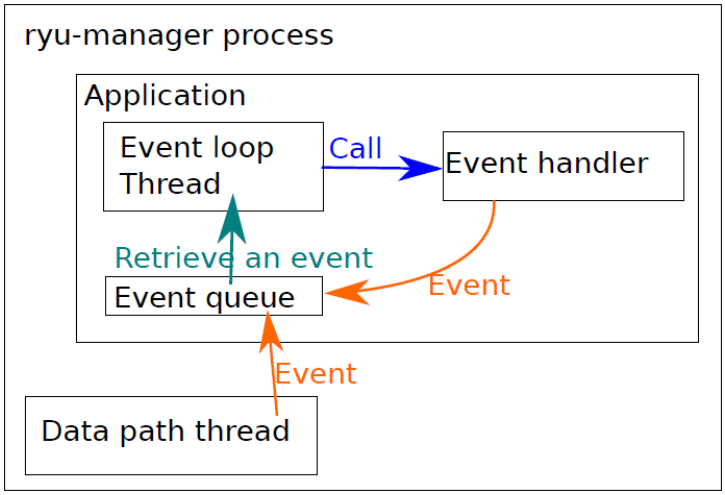


Figura 2.6: Modelo de programación de RYU
(Fuente: Ryu Project Team en [26])

Después de una breve reseña de cada uno de los diferentes controladores en la figura 2.7 se tiene un breve resumen comparativo de cada uno de ellos. Para este proyecto consideraremos lo siguiente: que sea código abierto, que utilice un lenguaje como java o

Python, pueda ejecutarse sobre Linux, tenga una interface web, soporte OpenFlow 1.3 y tenga buena documentación.

COMPARACION DE CONTROLADORES SDN						
	NOX	POX	Beacon	Floodlight	ODL	RYU
Licencia	GPL 3.0	Apache 2.0	GPL 2.0	Apache 2.0	EPL 1.0	Apache 2.0
Arquitectura	Centralizada	Centralizada	Centralizada	Centralizada	Distribuida Plana	Centralizada
Lenguaje	C++	Python	Java	Java	Java	Pyhton
Plataformas	Linux	Linux, MacOS, Windows	Linux, MacOS, Windows	Linux, MacOS, Windows	Linux, MacOS, Windows	Linux, MacOS
Codigo Abierto	SI	SI	SI	SI	SI	SI
Interfaz Gráfica	Web UI	CLI, GUI	CLI, Web UI	CLI, Web UI	Web	Web
North Bound API	ad-hoc	ad-hoc	ad-hoc	REST, Java, Quantum	REST, REST CONF, XMPP, NETCONF	REST
Soutth Bound API	Openflow 1.0	Openflow 1.0	Openflow 1.0	Openflow 1.0, 1.3	Openflow 1.0, 1.3	Openflow 1.0, 1.3, 1.5
Documentacion	Limitada	Limitada	Suficiente	Buena	Buena	Buena

Figura 2.7: Comparación de controladores SDN
(Fuente: Zhu L., Karim MM., Sharif K., Fan Li, Du X., & Guizani M. [30])

Del grafico podemos ver que ODL, FloodLight y RYU cumplen con los requisitos. Sin embargo, seleccionaremos para este proyecto como controlador a RYU; se prefiere usar Python por ser run lenguaje que se está posicionado cada vez más en el ámbito de la programabilidad en la red.

2.3. OpenFlow

Para poder realizar el despliegue de una implementación práctica de SDN es necesario cumplir dos requerimientos: primero todos los dispositivos de reenvío de paquetes deben compartir una misma arquitectura lógica, sin importar de que fabricante provengan; segundo, es necesario un protocolo estándar y seguro en el SBI que comunique el controlador y los dispositivos. Ambos requerimientos son cubiertos por OpenFlow (ver figura 2.12) que es tanto un protocolo que actúa como interfaz entre el controlador y los dispositivos de red; así como una especificación de la estructura lógica de las funciones de conmutación en la red.

OpenFlow fue creado como parte de un grupo de investigación de redes en la Universidad de Stanford, su propósito original era permitir la creación de protocolos experimentales que puedan utilizarse para la investigación. A partir de esta idea OpenFlow evolucionó como un protocolo capaz de reemplazar a los protocolos de capa 02 y 03 en los dispositivos comerciales de ruteo y conmutación. En 2011 la ONF fue creada para estandarizar, comercializar y promover el uso de OpenFlow en redes en producción.

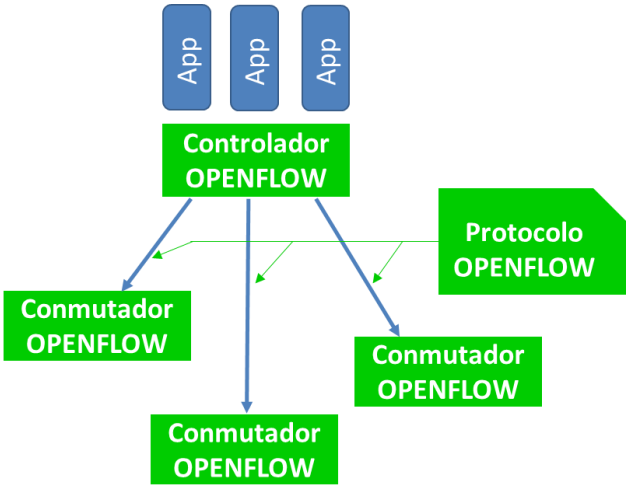


Figura 2.8: OpenFlow

La ONF define OpenFlow en el documento “OpenFlow Switch”, en la versión 1.3.5 de este documento publicada en 2015 podemos identificar 3 componentes: El controlador OpenFlow, el conmutador OpenFlow y el protocolo OpenFlow. En la figura 2.8 vemos cada uno de estos componentes en el contexto de la arquitectura SDN.

2.3.1. Controlador OpenFlow

Como ya se ha descrito en el punto 2.2, un controlador es el componente SDN que se encarga de gestionar la red recibiendo paquetes de información y enviando paquetes con órdenes que permitan actualizar las tablas de flujo. Un controlador OpenFlow, es un controlador que utiliza el protocolo OpenFlow para comunicarse en el SBI con los dispositivos de red, en este caso, con conmutadores OpenFlow.

2.3.2. Conmutador OpenFlow

Como podemos observar en la figura 2.9 un conmutador OpenFlow consiste en una o más tablas de flujo (*flow tables*) y una tabla de grupo (*group table*), las que ejecutan las

búsquedas y reenvío de paquetes; y uno o más canales OpenFlow (*OpenFlow Channel*) hacia uno o más controladores externos.

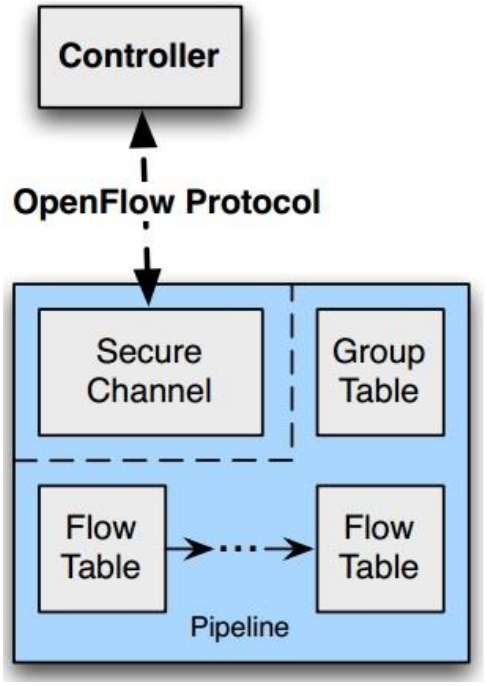


Figura 2.9: Arquitectura de un conmutador OpenFlow 1.3.0 (Fuente: ONF en [19])

Los conmutadores OpenFlow pueden ser puros o híbridos:

- Un conmutador OpenFlow puro tiene de manera predeterminada instalado el software que soporta OpenFlow y no da soporte a las capas dos y tres de OSI. No mantiene ninguna característica de un conmutador tradicional, ni control individual, depende exclusivamente de las decisiones enviadas por el controlador OpenFlow.
- Un conmutador híbrido utiliza OpenFlow además de las funciones tradicionales de conmutación. La mayoría de los conmutadores comerciales disponibles en la actualidad son híbridos, incluyen un conmutador lógico OpenFlow para poder ser utilizado dentro de una SDN.

Cada tabla de flujo en el conmutador mantiene un conjunto de entradas de flujo (*flow entries*) que están formadas por:

1. Campo de cabecera, con información de la cabecera del paquete, puerto de ingreso y metadata. Se utiliza para comparar los paquetes de entrada.

2. Contadores, utilizados para recolectar estadísticas de un flujo particular tal como el número de paquetes recibidos, numero de bytes y duración de cada flujo.
3. Un conjunto de instrucciones, que pueden ser aplicadas después de una coincidencia, que indique como se deben manejar los paquetes que cumplen la coincidencia. Por ejemplo, la acción puede ser reenviar el paquete a un puerto específico.

El controlador gestiona el conmutador a través del protocolo OpenFlow. Utilizando este protocolo, el controlador puede agregar, actualizar y eliminar entradas de flujo, tanto de manera reactiva (en respuesta a los paquetes) como proactiva.

En la figura 2.9 vemos los componentes básicos de un conmutador:

- Puertos OpenFLow, son interfaces de red utilizadas para pasar paquetes entre el proceso OpenFlow y el resto de la red.
- Tablas OpenFlow, las tablas de flujo indican el comportamiento para cada tipo de tráfico.
- La tabla de grupo, contienen entradas por cada grupo, en cada una de estas entradas hay un paquete de acciones que deben seguirse en función de cada tipo de grupo.
- Canal OpenFlow, es un canal seguro que permite utilizar el protocolo OpenFlow para comunicarse con el controlador.

De esta manera los diseñadores de conmutadores son libres de implementar de la manera que crean conveniente el núcleo de conmutación, mientras se les provea de las instrucciones semánticas correctas. Por ejemplo: mientras un flujo utiliza todo un grupo para reenviar a múltiples puertos, el diseñador podría implementarlo como una simple mascara de bits en la tabla de reenvío; otro ejemplo es la comparación, la secuencia de procesamiento mostrada en la figura 2.10 puede ser implementada en diferentes tablas en hardware.

2.3.2.1. Puertos OpenFlow

Los puertos OpenFlow son interfaces de red que permiten intercambiar paquetes entre el procesamiento OpenFlow y el resto de la red. Un conmutador OpenFlow habilita un

numero de puertos para el procesamiento OpenFlow, el conjunto de puertos no necesariamente es idéntico al conjunto de interfaces de red provistas por el hardware.

Los paquetes OpenFlow son recibidos en un puerto de entrada (*ingress port*) por la secuencia de procesamiento que posiblemente lo reenviara hacia un puerto de salida (*output port*). Un conmutador OpenFlow soporta 3 tipos de puertos:

- Puertos físicos, son los puertos definidos por el conmutador que corresponden a las interfaces de hardware del conmutador.
- Puertos lógicos, son puertos que no corresponden directamente a interfaces de hardware del conmutador. Los puertos lógicos son una abstracción de alto nivel y son definidos por el conmutador por métodos que no son de OpenFlow, por ejemplo: grupos de *link aggregation*, túneles, interfaces de loopback, etc.
- Puertos reservados, son definidos dentro de la especificación OpenFlow, estos especifican acciones genéricas de reenvío como enviar al controlador, hacer una inundación o reenviar utilizando métodos que no son de OpenFlow (como el reenvío tradicional realizado por el conmutador).

2.3.2.2. Tablas OpenFlow

La secuencia de procesamiento OpenFlow de cada conmutador lógico OpenFlow contiene una o más tablas de flujo, cada tabla de flujo contiene una serie de entradas de flujo. Como vemos en la figura 2.10, la secuencia de procesamiento OpenFlow define como los flujos de paquetes interactúan con las tablas de flujo. Un conmutador OpenFlow debe tener al menos una tabla de flujo de ingreso, aunque opcionalmente puede tener más tablas. Cada tabla de flujo debe estar numerada empezando por 0, para que puedan ser atravesadas por los paquetes; la secuencia de procesamiento se ejecuta en 2 etapas, procesamiento de ingreso (*ingress processing*) y procesamiento de salida (*egress processing*)

La secuencia de procesamiento siempre inicia en la primera tabla de ingreso, los paquetes son comparados primero con las entradas de flujo de la tabla 0, dependiendo del resultado el paquete puede ser comparado a través de otras tablas de flujo de ingreso. Si la salida de una tabla de flujo de ingreso es reenviar el paquete a un puerto de salida,

puede ejecutarse un procesamiento de salida en el contexto del puerto correspondiente; sin embargo, la ejecución de un procesamiento de salida es opcional.

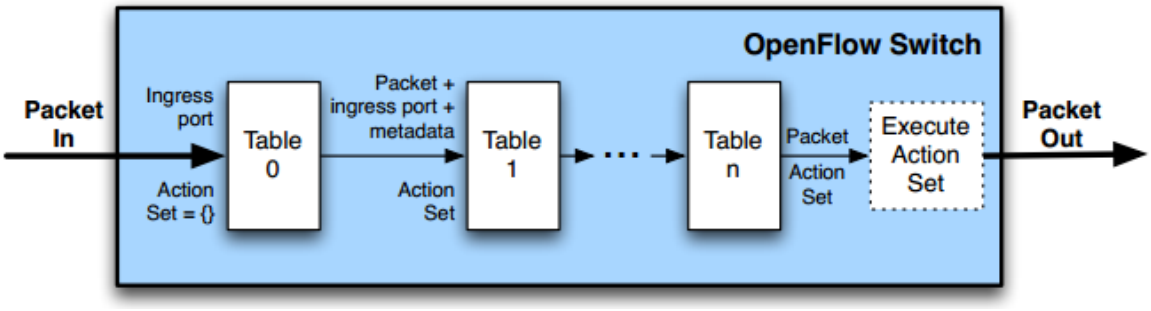


Figura 2.10: Flujo de un paquete a través de la secuencia de procesamiento de OpenFlow 1.3.0 (Fuente: ONF en [19])

Cuando un paquete es procesado por una tabla de flujo, este se compara con las entradas de flujo; si se encuentra una coincidencia las instrucciones incluidas en esa entrada son ejecutadas. Estas instrucciones podrían enviar explícitamente al paquete a ser comparado con otra tabla de flujo donde se repetirá el mismo proceso, o como ya se mencionó podría ser enviado a un puerto de salida.

Cada entrada de flujo consiste en campos de comparación, contadores y un conjunto de instrucciones que se deben aplicar a los paquetes, si coinciden con alguna política. En la figura 2.10 podemos observar el paso a través de la secuencia de procesamiento; la coincidencia empieza en la primera tabla de flujo y continua con las siguientes, los paquetes son comparados en orden de prioridad utilizando la primera entrada en cada tabla. Si se encuentra una entrada que coincide las instrucciones asociadas con ese flujo específico son ejecutadas. Si no hay coincidencia la salida depende de la configuración del conmutador: el paquete puede ser reenviado hacia el controlador a través del canal OpenFlow, descartado o puede continuar a la siguiente tabla de flujo. Las entradas de flujo pueden reenviar los paquetes a un puerto, normalmente es un puerto físico, pero también podría ser un puerto virtual definido en el conmutador o un puerto reservado.

En la tabla de flujos, cada entrada de flujo se asocia con una acción específica:

- Enviar el paquete por un puerto específico para que sea reenviado a través de la red.
- Enviar paquetes al controlador cuando se recibe un paquete que no coincide con ninguna tabla de flujo. El3 controlador decide como se llenará esa tabla de flujo.

- Encapsula y envía paquetes al controlador que pertenecen a un flujo de datos determinado. Esta acción se usa en el primer paquete de un nuevo flujo, para determinar si se añade o no a la tabla de flujos.
- Eliminar los paquetes pertenecientes a un flujo de tráfico, ya sea por motivos de seguridad u otras razones.

2.3.2.3. Canal OpenFlow

El canal OpenFlow actúa como una interfaz para conectar a cada conmutador lógico OpenFlow con uno o más controladores OpenFlow. A través de este canal el controlador puede configurar y gestionar el conmutador, recibiendo eventos y enviando paquetes hacia los conmutadores. Esta interfaz puede soportar un solo canal de control OpenFlow hacia un solo controlador o también múltiples canales hacia múltiples controladores que comparten la gestión de un conmutador. La implementación de la interfaz entre controlador y conmutador es específica, sin embargo, todos los mensajes deben ser enviados siguiendo el formato de OpenFlow. Este canal normalmente se encripta utilizando TLS, aunque puede ser utilizado directamente sobre TCP.

2.3.3. Protocolo OpenFlow

OpenFlow utiliza mensajes cuando se comunica con otros dispositivos de la red, estos mensajes son enviados a través del canal OpenFlow. Dentro del canal, soporta el envío de 3 tipos de mensajes:

- Controlador-a-conmutador (controller-to-switch), se inician por el controlador y se utilizan para gestionar o inspeccionar directamente el estado del conmutador. Estos mensajes los vemos en la tabla 2.1.
- Asíncronos son iniciados por el conmutador y se usan para actualizar al controlador sobre eventos en la red o cambios en el estado del conmutador. Estos mensajes los vemos en la tabla 2.2.
- Simétricos se inician por el controlador o el conmutador y se envían sin necesidad de una solicitud. Estos mensajes los vemos en la tabla 2.2.

Tabla 2.1: Mensajes OpenFlow: Conmutador a Controlador
(Fuente: ONF en [19])

Mensaje	Descripción
Commutador a Controlador	
Features	*Request the capabilities of a switch. Switch responds with a feature reply that specifies its capabilities.*
Configuration	*Set and query configuration parameters. Switch responds with parameter settings.*
Modify-State	*Add, delete, and modify flow/group entries and set switch port properties.*
Read-State	*Collect information from switch, such as current configuration, statistics, and capabilities.*
Packet-out	*Direct packet to a specified port on the switch.*
Asynchronous-Configuration	*Set filter on asynchronous messages or query that filter. Useful when switch connects to multiple controllers.*

Tabla 2.2: Mensajes OpenFlow: Asíncronos y Síncronos
(Fuente: ONF en [19])

Mensaje	Descripción
Asíncronos	
Packet-in	*Transfer packet to controller.*
Flow-Removed	*Inform the controller about the removal of a flow entry from a flow table.*
Port-Status	*Inform the controller of a change on a port.*
Error	*Notify controller of error or problem condition.*
Síncronos	
Hello	*Exchanged between the switch and controller upon connection startup.*
Echo	*Echo request/reply messages can be sent from either the switch or the controller, and they must return an echo reply.*
Experimenter	*For additional functions.*

2.3.3.1. Mensajes Controlador-a-Conmutador

Son mensajes que se originan en el controlador y no siempre requieren que el controlador responda. Entre ellos tenemos:

- *Features*: solicita las características del conmutador. Se usa normalmente durante el establecimiento de una conexión.
- *Configuration*: el controlador puede consultar o determinar los parámetros de configuración del conmutador. El conmutador solo responde a consultas de parte del controlador.
- *Modify-State*: gestionan los estados del conmutador, como agregar o eliminar flujos, cambiar el estado de un puerto especifico.
- *Read-State*: solicita estadísticas del conmutador tales como configuración actual, estadísticas y capacidades
- *Packet-Out*: se utiliza para enviar paquetes por un determinado puerto del conmutador y para reenviar paquetes recibidos a través de mensajes Packet-In. En su interior este paquete contiene el paquete que será reenviado y las acciones que se aplicarán al mismo.
- *Barrier*: el controlador lo usa para asegurarse que las dependencias de un mensaje se han cumplido o para recibir notificaciones cuando se finaliza una operación.
- *Role-Request*: el controlador lo utiliza para establecer el rol de su canal OpenFlow, configurar o consultar el ID de controlado. Este mensaje es muy útil cuando un conmutador se conecta a varios controladores diferentes.
- *Asynchronus-Configuration*: con estos mensajes el controlador asigna filtros adicionales en los mensajes asíncronos que desea recibir a través del canal OpenFlow o hace consultas a los mismos. Este mensaje es muy útil cuando un conmutador se conecta a varios controladores diferentes.

2.3.3.2. Mensajes Asíncronos

Son enviados por el conmutador sin la necesidad de ser solicitados por el controlador. Los conmutadores envían estos mensajes al controlador para indicar llegada de paquetes, cambios de estado del conmutador o errores. Los 4 mensajes asíncronos principales son:

- *Packet_In*: se envía cuando el conmutador no tiene una entrada en su tabla de flujos que concuerde con el paquete entrante. El controlador procesa el paquete y responde con un mensaje *Packet-Out*.
- *Flow-Removed*: se envía cuando vence el tiempo de inactividad o el tiempo de vida de un flujo. Puede ser enviado tanto por el controlador como por el conmutador.
- *Port-Status*: Se envía del conmutador al controlador cuando la configuración de un puerto cambia.
- *Controller-Status*: informa al controlador cuando hay cambios en el estatus del canal OpenFlow. El conmutador envía estos mensajes a todos los controladores cuando el estado de cualquier canal OpenFLow cambia. Esto permite disminuir las fallas de procesamiento cuando los controladores pierden la habilidad de comunicarse entre ellos.
- *Flow-Monitor*: informa a los controladores de los cambios en las tablas de flujo. Un controlador debe definir un conjunto de monitores para rastrear cambios en las tablas de flujo.

2.3.3.3. Mensajes Síncronos

Son enviados sin necesidad de una solicitud previa. Entre ellos tenemos:

- *Hello*: estos mensajes se intercambian entre los conmutadores y el controlador al momento del establecimiento de la conexión.
- *Echo (Request/Reply*): se utilizan para medir la latencia, el ancho de banda o verificar que un dispositivo esté activo.
- *Experimenter*: proveen un mecanismo estándar para que los conmutadores OpenFlow provean funcionalidades adicionales. En general este mensaje ha sido planeado para futuras versiones de OpenFlow.
- *Error*: El conmutador notifica al controlador si existen problemas con estos mensajes. Estos mensajes son usados mayormente por el conmutador para indicar fallas de las solicitudes iniciadas por el controlador.

2.3.4. Manejo de Mensajes

El protocolo OpenFlow provee una entrega y procesamiento confiable de mensajes, sin embargo, no provee confirmaciones automáticas o asegura el procesamiento ordenado de los mensajes.

- Entrega de Mensajes. La entrega de los mensajes es garantizada, a menos que exista fallas en la conexión en cuyo caso el controlador no asume ningún estado para el conmutador.
- Procesamiento de Mensajes. Los conmutadores deben procesar todos los mensajes recibidos desde el controlador, y si es necesario generar una respuesta. Si un conmutador no puede completar el procesamiento de un mensaje recibido desde el controlador, debe enviar un mensaje de error. En el caso de los mensajes *packet-out* el procesamiento completo del mensaje no garantiza que el paquete incluido en el mensaje abandone el conmutador. El paquete incluido debe ser eliminado debido a congestión en el conmutador, aplicar alguna política de QoS o ser enviado a un puerto bloqueado o invalido.
- Agrupación de Mensajes: el controlador puede agrupar mensajes que sean relacionados, el conjunto de mensajes en el grupo es aplicado como una unidad, se procesan juntos, y si paree algún error ningún mensaje es aplicado. Si el grupo se configura como ordenado, entonces los mensajes se aplican en el orden específico.
- Ordenamiento de los Mensajes. Mantener el orden de los mensajes se puede asegurar mediante el uso de mensajes de barrera. En la ausencia de mensajes de barrera los conmutadores reordenan los mensajes de manera arbitraria para maximizar el desempeño, debido a que los controladores no dependen de un orden específico al momento del procesamiento. Los mensajes no deben ser reordenados cruzando un mensaje de barrera y el mensaje de barrera debe ser procesado solo cuando todos los mensajes anteriores han sido procesados. Para ser más precisos seguimos lo siguiente:

    1. Los mensajes anteriores al mensaje de barrera deben ser procesados por completo antes del mensaje de barrera. Esto incluye enviar cualquier respuesta, resultado o errores.
    2. El mensaje de barrera es procesado y se envía la respuesta correspondiente.
    3. Los mensajes siguientes al mensaje de barrera recién son procesados.

2.4. Mininet

Mininet es una herramienta muy útil en la emulación de SDN y será la herramienta que utilizaremos en este proyecto por lo que comentamos brevemente sobre su funcionamiento. Mininet es mantenido en el sitio web: mininet.org donde se puede encontrar documentación y tutoriales.

Según mininet.org en [16]: “Mininet es un emulador de red, que proporciona un marco de pruebas y un entorno de desarrollo virtual para las redes definidas por software (SDN). Los desarrollos realizados en mininet pueden desplegarse con facilidad a una red real. Provee por lo tanto una plataforma económica, para crear hosts virtuales y otros elementos de red como switches, controladores y enlaces.”

De acuerdo con la ONF una red mininet consiste en:

- Hosts aislados, un grupo de procesos a nivel de usuario que se mueven dentro del espacio de nombres de una red, que proveen interfaces exclusivas, puertos y tablas de ruteo.
- Enlaces emulados, el componente Linux Trafic Control asegura que la tasa de datos de cada enlace modele el tráfico, de acuerdo con su configuración. Cada host emulado tiene su propia interfaz ethernet virtual.
- Conmutadores emulados, el puente usado por defecto por Linux o el OpenVSwitch que se ejecuta en modo kernel, se encarga de conmutar los paquetes a través de las interfaces. Conmutadores y ruteadores pueden correr en el kernel o en el espacio de usuario.

Las estaciones de trabajo mininet corren Linux como sistema operativo y sus conmutadores soportan OpenFlow para manejar la SDN. Mininet soporta investigación, desarrollo, aprendizaje, prototipado, pruebas, depuración de errores y cualquier otra tarea que pueda beneficiarse de contar con red experimental completa en una PC o laptop.

Cuando se crea una red virtual con mininet, como se ve en la figura 2.11, el controlador utiliza la dirección de loopback (127.0.0.1) y el puerto 6533 para establecer el socket de escucha con los conmutadores. Mientras el conmutador escuchara el envío de mensajes

a través del comando dpctl, desde el espacio del usuario, en la dirección de loopback (127.0.0.1) y el puerto 6534.

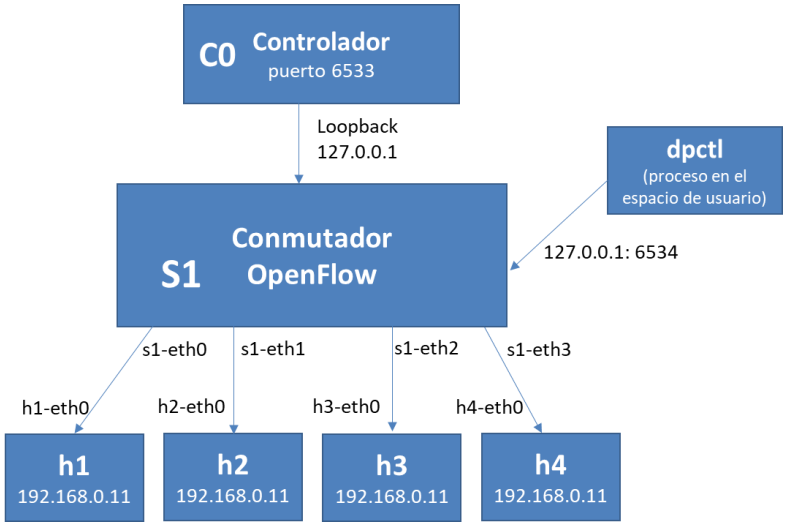


Figura 2.11: Red Virtual en Mininet.

Algunas de sus características:

- Provee un entorno de pruebas para el desarrollo de aplicaciones OpenFlow simple y de bajo costo.
- Permite trabajar a múltiples desarrolladores concurrentes de manera independiente en la misma topología.
- Incluye una interfaz de línea de comandos (CLI) que es independiente de la topología y OpenFlow, que permite la depuración y la ejecución de pruebas a lo largo de la red.
- Soporta topologías personalizadas e incluye un conjunto básico de topologías parametrizadas.
- Provee un API de Python para la creación y experimentación de la red.

Las redes mininet ejecutan código real que incluye aplicaciones estándar de UNIX/LINUX, así como un kernel real de Linux y la pila de protocolos de red. Por lo tanto, el código que se desarrolla y se prueba en mininet para un controlador OpenFlow, conmutador o computadora; puede ser movido a un sistema real con mínimos cambios, para pruebas en el mundo real, evaluación de desempeño y despliegue en producción. Esto significa que un diseño que trabaja en mininet puede ser movido directamente a conmutadores reales en una red física sin mayores cambios.

La virtualización que usa se basa en procesos y usa espacios de nombres, que permiten manejar procesos individuales, con interfaces, tablas de ruteo y tablas ARP separadas. Como desventaja, mininet depende del kernel de Linux, por lo que no puede ser implementado sobre otro sistema operativo, aunque actualmente se pretende dar soporte al uso de esta plataforma en otros sistemas operativos.

Mininet se destaca por ser una herramienta que combina la rapidez y la escalabilidad. Además, es una herramienta de fácil instalación que cuenta con dos opciones:

1) A través de una máquina virtual con todo instalado y configurado.
2) Instalación directa del código fuente, de preferencia sobre un servidor Ubuntu.
3) Y finalmente si se está ejecutando un servidor Ubuntu, también es posible instalarla desde paquetes pre-compilados.

Se escogió la instalación con VM por ser la más practica y rápida de utilizar.

2.4.1. Manejando Mensajes con Mininet

Mininet implementa el comando dpctl que es una utilidad de gestión que permite el control sobre un conmutador OpenFlow. Con dpctl podemos agregar flujos a la tabla de flujos, consultar por el estado y las características del conmutador, y editar otras configuraciones.

La sintaxis del comando dpctl add-flow se detalla a continuación:

#dpctl add-flow [protocolo]:[ip_conmutador]:[puerto] nw_proto=[protocolo_filtrado], nw_src=[ip_origen], nw_dst=[ip_destino], dl_src=[mac_origen], dl_dst=[mac_destino], tp_src=[puerto_origina], tp_dst=[puerto_destino], in_port=[#puerto_entrada], idle_timeout=[tiempo_expiracion_inactividad], hard_timeout[tiempo _expiracion], dl_vlan=[VLAN],priority[prioridad], actions=[output:#puerto_salida]/[normal]/[flood]/[enqueue:puerto:id]/[all]/[controller]/[mod_vlan_id:id]

En donde:

- [protocolo]: es el protocolo con el cual se realizará la comunicación, el protocolo más utilizado es TCP, sin embargo, si se requiere una comunicación segura se puede usar SSL.
- [ip_conmutador]: es la dirección IP asignada al conmutador.
- [puerto]: define el puerto en el que el controlador escucha los mensajes OpenFlow.
- [protocolo_filtrado]: se usa para filtrar los paquetes de un determinado protocolo, como por ejemplo IP, HTTP, etc.
- [ip_origen] e [ip_destino]: se usan para caracterizar al flujo, es decir, clasificarlo de acuerdo con la dirección IP que origina la transmisión y la dirección de destino. Se puede usar una dirección de red con máscara para especificar un rango de direcciones.
- [mac_origen] y [mac_destino]: son las direcciones MAC de los dispositivos ya sea de origen o de destino de la comunicación.
- [puerto_origen] y [puerto_destino]: son los identificadores de número de puerto TCP o UDP ya sea de origen o de destino de la comunicación.
- [#puerto_entrada]: es el puerto por el que ingresan los paquetes de un determinado flujo, y servirá para definir y caracterizar el flujo.
- [tiempo_expiracion_inactividad]: es el tiempo en el que un determinado flujo se mantiene en la tabla de flujos antes de ser eliminado por inactividad. Si no se envían paquetes de este flujo en este periodo, el flujo se elimina de la tabla. Se usa cero (0) para flujos permanentes.
- [tiempo_expiracion]: es el tiempo en el que un determinado flujo se mantiene en la tabla de flujos antes de ser eliminado ya sea que hayan llegado paquetes de ese flujo o no. Se usa cero (0) para flujos permanentes.
- [VLAN]: es la etiqueta de una determinada VLAN que el tráfico entrante podría tener
- [prioridad]: define la prioridad con la que se tratará el paquete.

Opciones de tratamiento de paquetes (*actions*):

- [output: #puerto_salida]: es el puerto al cual se desea redirigir el tráfico. Por ejemplo, el flujo llegó al puerto denominado eth0 y se desea reenviarlo a un equipo conectado al puerto eth2.

- [normal]: el conmutador le da un comportamiento normal de capa 2/3 al paquete.
- [*flood*]: Envía los paquetes a todas las interfaces del conmutador, excepto por la que el paquete entro y las que tienen la opción *flooding* deshabilitada.
- [enqueue:puerto:id]: especifica que se pondrá en cola al paquete. Como parámetros se define el puerto físico de salida y un identificador para monitoreo del paquete.
- [all]: envía el paquete a todas las interfaces (puertos), menos a la interfaz por la que el paquete entró.
- [controller]: envía el paquete al controlador, para que este decida qué hacer con él.
- [mod_vlan_id:id]: sirve para modificar el identificador de la VLAN cuando se redirige el paquete hacia la interfaz de salida.

Otro comando importante es mod-port, utilizado para modificar la configuración de los puertos físicos del conmutador OpenFlow. Aquí se detallan sus parámetros:

#dpctl mod-port [protocolo]:[ip_controlador]:[puerto] [puerto_fisico] [estado]

En donde:

- [puerto_fisico]: es el identificador de puerto físico del conmutador.
- [estado]: es el comportamiento que tiene el puerto, y puede ser:

  down apaga el puerto.
  up enciende el puerto.
  flood el puerto participa en caso de una inundación de paquetes (*flooding*).
  noflood el puerto no participa en caso de una inundación de paquetes (flooding)

2.4.2. Porque usar Mininet

Mininet combina muchas de las mejores características de los emuladores, hardware para pruebas y simuladores.

Comparado con un sistema virtualizado completo:

- Arranca con rapidez, segundos en vez de minutos.
- Tiene gran escalamiento, cientos de computadoras y conmutadores.
- Provee más ancho de banda, típicamente 2Gbps en un hardware moderado.

- Se instala con facilidad, hay una máquina virtual pre empacada que corre en VMWare o VirtualBox con OpenFlow instalado.

Comparado con hardware de pruebas:

- Es de muy bajo costo y siempre está disponible.
- Se reconfigura y se puede reiniciar con rapidez.

Comparado con simuladores:

- Ejecuta código real, incluyendo código de aplicaciones, código de kernel y código del plano de control (tanto código del controlador OpenFlow, como código del conmutador).
- Se conecta con facilidad a redes reales.
- Ofrece un desempeño interactivo.

2.5. Topologías de red para Centros de Datos

Los centros de datos almacenan y procesan grandes cantidades de información; actualmente soportan a los servicios de computación en la nube; la elección correcta de la topología de red cumple un rol importante para conseguir el desempeño esperado.

Liu Y., Muppala J., Veeraraghavan M., Lin D., Hamdi M., en [14] definen como topología de red a los diferentes tipos de estructura en las que se organizan las conexiones entre los dispositivos de red para transmitir información entre ellos.

Como vemos en la figura 2.12 hay una gran variedad de topologías que pueden ser utilizadas para satisfacer diferentes necesidades de un centro de datos. Se dividen principalmente en Topologías Rígidas (Fixed Topologies) y Topologías Flexibles (Flexible Topologies). Dentro de las Topologías Rígidas son las Topologías de Árbol (*Tree-based topologies*) las que se utilizan ampliamente en las redes de los centros de datos. Por ejemplo, CISCO recomienda una topología multi capa basada en árbol como se muestra en la figura 2.13.

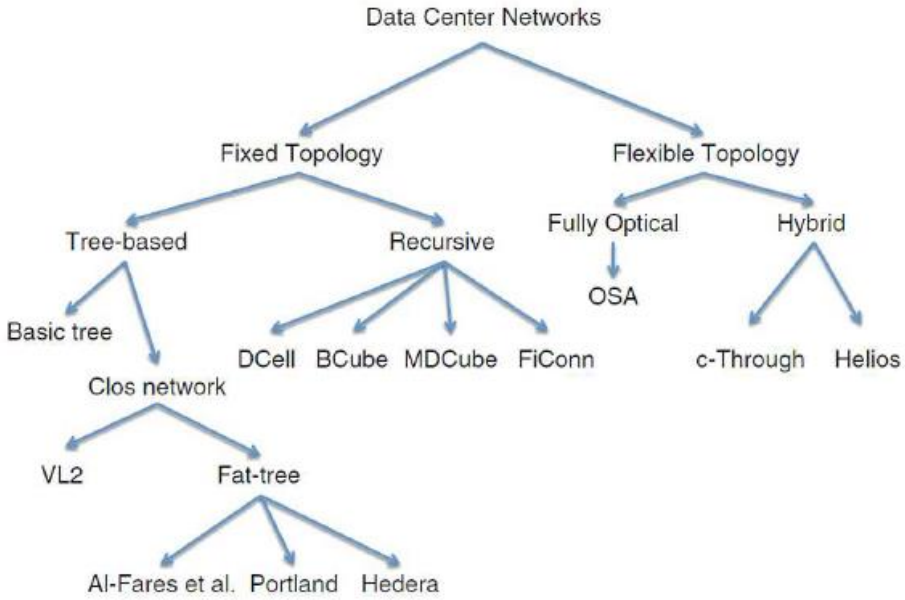


Figura 2.12: Taxonomía de las topologías de red para centros de datos. (Fuente: Liu et al en [14])

2.5.1. Basic Tree

La topología *BasicTree* está conformada por dos o tres niveles de conmutadores (o ruteadores), con los servidores como hojas. En una topología de 3 niveles: la raíz del árbol es la capa de core, en el medio esta la capa de agregación y en el borde la capa de acceso a los servidores. No hay enlaces en entre los conmutadores de la misma jerarquía. Esta es una topología ampliamente utilizada en los centros de datos comerciales y no tiene una definición única ya que depende las especificaciones de los equipos comerciales, así como de las decisiones de diseño utilizadas.

CISCO recomienda esta topología, ver figura 2.13, conmutadores Top of Rack conectados a conmutadores de acceso, los que a su vez se conectan con conmutadores de agregación y estos a conmutadores de core.

Couto R., Secci S., Mitre M., Kosmalski L. en [5] mencionan que en una arquitectura de 3 capas:

> La capa de core es compuesta por dos conmutadores conectados directamente entre ellos (actúan como puertas de enlace para el centro de datos); cada uno de ellos se conecta a todos los conmutadores de agregación. Los conmutadores de agregación se suelen conectar en parejas, en cada pareja los

conmutadores se conectan uno al otro y cada conmutador de agregación está conectado al mismo grupo de n_a conmutadores de acceso. Cada conmutador de acceso tiene n_e puertos para conectarse directamente a los servidores. (2015, p.4)

Por lo tanto, podemos ver que:

- Cada par de conmutadores de agregación provee conectividad a $n_a \times n_e$ servidores.
- Se necesitan $\frac{|S|}{n_a \times n_e}$ pares de conmutadores de agregación para atender a S servidores.

Como ejemplo podemos ver en la figura 2.13 una topología conformada por 16 servidores con $n_a = 4$ y $n_e = 2$.

Podemos observar además que, en esta topología, los conmutadores a medida que se encuentran en la parte superior de la jerarquía manejan más tráfico, por lo que requieren mayores funcionalidades de tolerancia a fallos. Esta topología es escalable y permite dependiendo de la capa a la que pertenece un conmutador definir diferentes funcionalidades y características; sin embargo, esto influye también en el costo, ya que los conmutadores aumentan en funcionalidades y costo a medida que se ubican en una capa superior de la jerarquía.

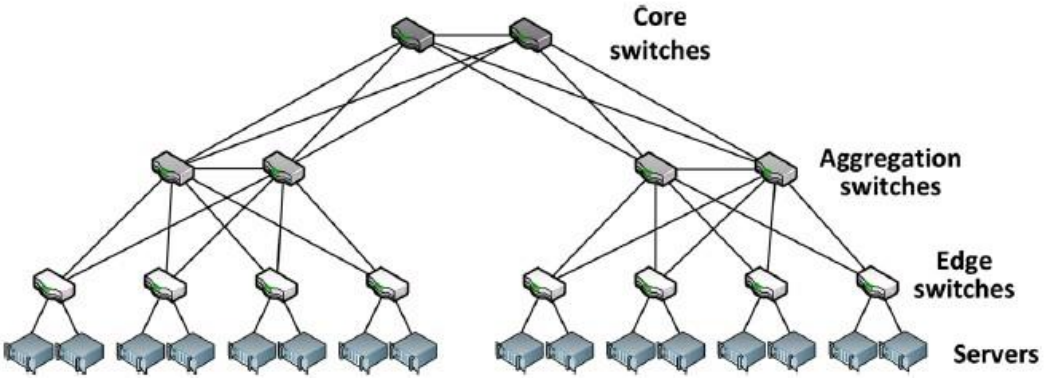


Figura 2.13: Topología basic-tree de 3 capas: $n_e = 2$ y $n_a = 4$.
(Fuente: Cuoto et al en [5])

2.5.2. Fat Tree

Al-Fares et al en [1], nos presenta la primera topología de red para centros de datos basada en Fat-Tree, en su artículo “A Scalable, Commodity Data Center Network Architecture” presentado en el SIGCOMM 2008: “Mostraremos que, interconectado conmutadores comunes en una arquitectura Fat-Tree, podemos alcanzar la completa bisección del ancho de banda de los clústeres conformados por decenas de miles de nodos” (Al-Fares et al, 2008).

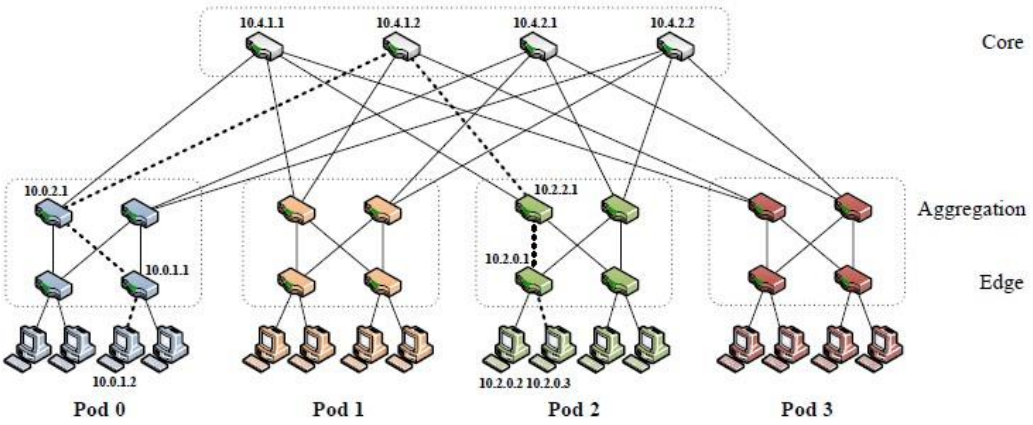


Figura 2.14: Topología Fat-Tree con $n = 4$.
(Fuente: Al-Fares et al en [1])

En este enfoque, como se observa en la figura 2.19 cada conmutador de $n - puertos$ en la capa de acceso se conecta a $(\frac{n}{2})$ servidores. Los $(\frac{n}{2})$ puertos restantes se conectan a $(\frac{n}{2})$ conmutadores en la capa de agregación. Los $(\frac{n}{2})$ conmutadores de capa de agregación, los $(\frac{n}{2})$ conmutadores de capa de acceso y los servidores conectados a estos últimos forman una unidad básica del fat-Tree que se llama un “pod”. Al nivel de core, hay $(\frac{n}{2})^2$ conmutadores de $n - puertos$, cada uno conectado con cada uno de los $n - pods$. El nuero máximo de servidores que podemos conectar en un Fat-Tree con conmutadores de $n - puertos$ es de $\frac{n^3}{4}$.

La figura 2.19, nos muestra un ejemplo de un Fat-Tree con $n = 4$, en este caso el nuero máximo de servidores que podemos conectar es de $\frac{n^3}{4} = 16$. Cada conmutador de acceso utiliza $\left(\frac{n}{2}\right) = 2$ enlaces para conectarse a 2 servidores y $\left(\frac{n}{2}\right) = 2$ conmutadores de

agregación. Podemos observar como el número de enlaces entre conmutadores de diferentes capas aumenta exponencialmente a medida que subimos en la jerarquía.

A diferencia de la topología BasicTree, utilizada con frecuencia en las redes de los centros de datos comerciales, todos los niveles utilizan el mismo tipo de conmutador. No es necesario conmutadores de alto desempeño en la agregación o en el core; además que podemos observar que FatTree presenta mayor redundancia en los enlaces, garantizando el establecimiento de rutas de comunicación ante caídas de la red.

2.5.3. Elastic Tree

Como se acaba de ver en el punto anterior una topología de red FatTree, permite a los centros de datos reducir costos en equipamiento mientras se aumenta la redundancia de los enlaces en casos de fallas. Sin embargo, FatTree presenta una debilidad; al estar diseñado para soportar altas tasas de tráfico durante las horas pico y proveer tolerancia a fallos gracias a su configuración redundante, durante las horas normales de funcionamiento trabaja muy por debajo de su capacidad. El tráfico en los centros de datos varia constantemente durante el día, el mes o el año; por lo que no trabajar a su capacidad completa por periodos prolongados se traduce en un alto consumo de energía, convirtiéndolo en una solución no eficiente desde el punto de vista energético.

En el año 2010 Heller B., Seetharaman S., Mahadevan P., Yiakoumis Y, Sharma P., Banerjee S. & McKeown N., en su artículo ElasticTree: Saving Energy in Data Center Networks en [12]; proponen ElasticTree como una alternativa para ajustar de forma dinámica el consumo de energía de la red de un centro de datos en función del tráfico que la atraviesa.

ElasticTree busca optimizar el consumo de energía en una red, determinando a través de un algoritmo de optimización que elementos de la red deben estar activos en un momento dado (tanto enlaces como conmutadores) en función del tráfico que la atraviesa en ese instante. Como observamos en la figura 2.15 ElastricTree mantiene un subconjunto de elementos de la red del centro de datos activos para garantizar la conectividad entre los pods de tomando como base una topología FatTree.

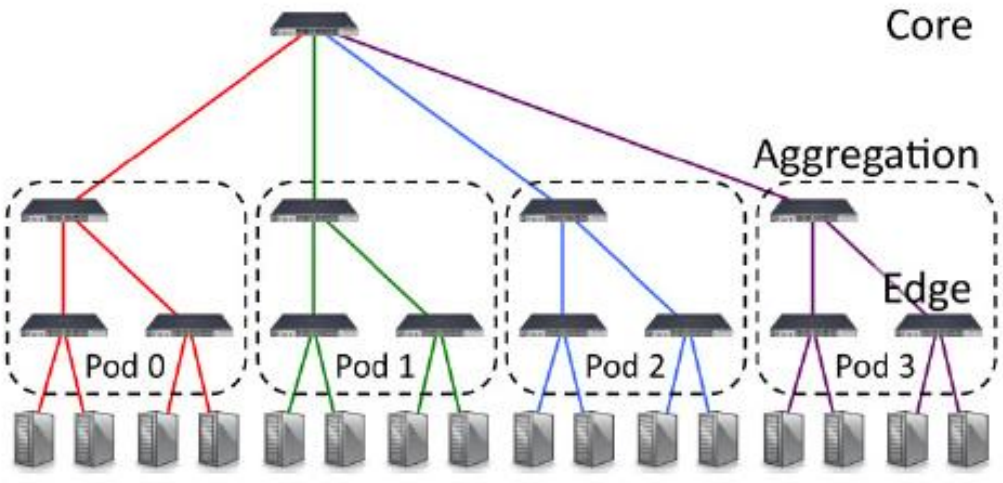


Figura 2.15: Un subconjunto de ElasticTree
(Fuente: Heller et al en [12])

Como vemos en la figura 2.16 los componentes son:

- El Optimizador, recibe como entrada la matriz de topología de la red, la matriz de tráfico de la red y el modelo de consumo de energía de los componentes de la red. Y se encarga de obtener el conjunto de componentes mínimos necesarios que deben estar activos para atender las condiciones de tráfico en ese momento.
- El Control de Energía, recibe como entrada el conjunto de componentes activos que fue seleccionado en el Optimizador, con esa información apaga y/o enciende puertos o conmutadores.
- El Enrutador, recibe como entrada el conjunto de componentes activos que fue seleccionado en el Optimizador, con esa información envía las rutas que deberán utilizar los conmutadores para reenviar los flujos de tráfico.

ElasticTree ha sido aplicado en Centros de Datos de gran envergadura, por ejemplo, en 2009 fue aplicado en Google a una aplicación de comercio electrónico en producción que tenía 292 servidores, donde se evidencio un ahorro de energía importante.

“El ahorro de energía depende del patrón de tráfico, … y del tamaño del centro de datos. Nuestros experimentos muestran en promedio un ahorro de energía entre el 25% al 40% es posible en un centro de datos” (Heller et al en [12]).

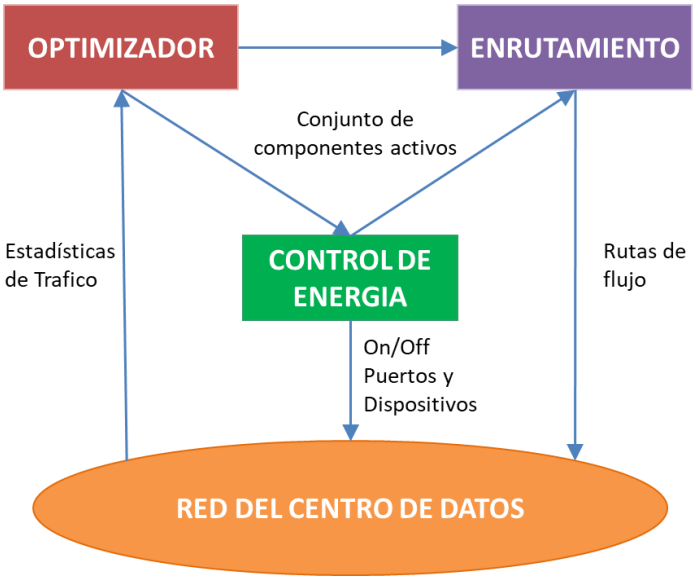


Figura 2.16: Componentes de control de ElasticTree
(Fuente: Adaptado de Heller et al en [12])

2.6. Modelo de Consumo de Energía

El modelo energético de la red se realizará considerando a Mahadevan P. et al en [15], donde la energía consumida por un conmutador depende de dos componentes:

- Un componente fijo en función del chasis del conmutador y las tarjetas de línea.
- Un componente variable en función del número de puertos activos, capacidad del puerto y su consumo cuando es utilizado.

Truong, D. -., Ouro, E., & Nguyen, T en su artículo Protected Elastic-tree topology for Data Center en [33], parten de esos dos principios anteriores y nos dan la fórmula:

$$P_{switch} = P_{chassis} + n_{linecards} \times P_{linecard} + \sum_{i} nport_{r_i} \times P_{r_i} \times F_u \quad (2.1)$$

- P_{switch} -> Energía consumida por el conmutador.
- P_{chasis} -> Energía consumida por el chasis del conmutador.
- $n_{linecards}$ -> Numero de tarjetas de línea.
- $P_{linecards}$ -> Energía consumida por una tarjeta de línea sin considerar el consumo por puerto.
- P_{ri} -> Energía consumida por puerto activo.
- $nport_{ri}$ - > número de puertos activos de tipo i (10Mbps, 100Mbps, 1Gbps)
- F_u -> Es un factor de escala que se utiliza por puerto.

Por motivos de facilidad se considerará a F_u con el mismo valor para todos los puertos de los conmutadores y con valor $F_u = 1$ para todos los puertos.

La Tabla 2.3 nos muestra un ejemplo del consumo de energía para 3 marcas diferentes de conmutadores de 48 puertos y diferentes configuraciones; la utilizaremos como referencia para calcular el modelo energético de la solución propuesta.

Tabla 2.3: Consumo de energía de 3 diferentes marcas de conmutadores. (Fuente: Heller et al en [12])

Puertos Habilitados	Trafico Por Puerto	Modelo A Potencia (W)	Modelo B Potencia (W)	Modelo C Potencia (W)
Ninguno	Ninguno	151	133	76
Todos	Ninguno	184	170	97
Todos	1 Gbps	195	175	102

Los resultados de esta tabla 2.3 son consolidados por Heller et al en [12] y tomando como fuente el trabajo de Mahadevan P. et al en [15], donde se realizan pruebas con diferentes marcas y modelos, haciendo pruebas de consumo de energía de manera experimental. Algunas de las conclusiones obtenidas en este trabajo:

- La relación entre la potencia real consumida por el dispositivo en promedio y su potencia nominal máxima varía ampliamente entre las diferentes familias de dispositivos. Por lo tanto, depender simplemente de la potencia nominal máxima puede sobrestimar enormemente la energía total consumida por estos equipos de red.
- La energía consumida por un conmutador se incrementa de manera lineal, de acuerdo con el número de tarjetas de línea conectadas, así como de puertos activos.
- La energía consumida por un conmutador es ampliamente independiente del tamaño del paquete para un rendimiento de tráfico fijo.

2.7. Algoritmos de Optimización

Para implementar el optimizador se puede utilizar varios tipos de algoritmo que permitan la elección del conjunto de componentes activos más eficiente energéticamente. Heller et al. en [12] analiza el uso de 3 algoritmos para Elasic Tree:

- El Modelo Formal, requiere ser programado en un lenguaje para modelado de optimización, como MathProg o GAMS y además requiere mucho más tiempo de cómputo para encontrar la solución óptima, sin embargo, garantiza encontrar la solución óptima para cualquier tipo de topología.
- GHA (*Greedy Heuristic Algorithm*), en este caso *Greedy Bin-Packing* es un algoritmo escalable a mayor número de hosts y con un tiempo computacional razonable. Sin embargo, por el tipo de estrategia que utiliza es posible que para ciertas matrices de tráfico no encuentre el óptimo deseado. Una ventaja es que calcula la solución de manera incremental lo que mejora su eficiencia computacional.
- *Topology-aware Heuristic*, este algoritmo mejora el tiempo de cómputo y es altamente escalable. Sin embargo, no computa las rutas de flujo completas más bien las divide y solo necesita como entrada el conteo de puertos.

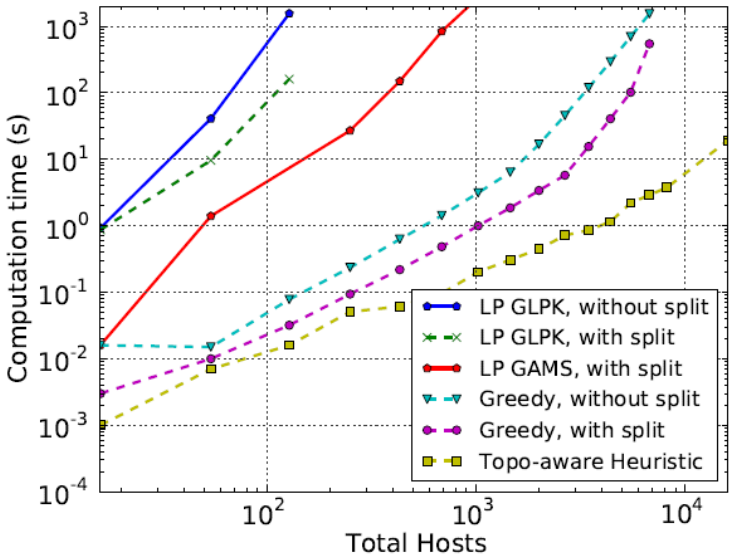


Figura 2.17: Tiempo de cómputo vs el tamaño de la red.
(Fuente: Heller et al en [12])

El resultado de este análisis es mostrado en la figura 2.17. Aquí podemos observar que el modelo formal alcanza resultados muy cercanos a optimo, pero para centros de datos grandes (por ejemplo, un FatTree con $n \geq 14$) el tiempo de cálculo es indeterminado. Para el caso del algoritmo GHA el tiempo mejora bastante en comparación al modelo formal, sin embargo, también se vuelve poco práctico para centros de datos muy grandes. El modelo heurístico es el que da mejores resultados en el caso de centros de datos

grandes, manteniendo una relación lineal entre el número de hosts y el tiempo que toma ejecutar el algoritmo.

De este punto concluimos que un algoritmo de tipo *Geedy* (Algoritmo voraz) es bastante aceptable para el numero de hosts que utilizaremos. Sin embargo, hay un algoritmo conocido como A*, que agrega una función adicional a la del algoritmo voraz para el cálculo de la función de evaluación f(n). Además, su facilidad de aplicación encaja con este proyecto, como será explicado más adelante en el capítulo 03 al momento de su aplicación.

2.7.1. Algoritmo A*

El algoritmo A*, se lee A Estrella (A Star), es un algoritmo de búsqueda informada que clasifica dentro de los algoritmos de búsqueda en grafos. A* permite encontrar el camino óptimo a través de su función de evaluación f(n). En 1968 fue presentado como una extensión del algoritmo de Djkstra por Peter E. Halt, Bertram Raphael y Nils J. Nilsson. A diferencia del algoritmo voraz la función de evaluación de A* incluye un componente adicional que evalúa el costo de la ruta recorrida hasta ese momento, lo que en las condiciones adecuadas permite encontrar la ruta optima y más rápido.

Principales características:

- Es un algoritmo considerado completo, si hay una solución posible, dará con ella.
- La complejidad computacional del algoritmo se basa en la calidad de la heurística utilizada. Si la función h(n) es mala la complejidad podría aumentar de manera exponencial, si la función h(n) es buena el algoritmo se ejecutará en tiempo lineal.
- La memoria requerida para ejecutar este algoritmo es alta, debido que debe almacenar los nodos posibles de cada estado. Hay otras versiones de A* que permiten mejorar esta característica.

El algoritmo A* calcula la solución óptima en considerando dos funciones: la función del costo de la distancia avanzada desde el punto de inicio g(n) y a una función heurística que mide la distancia del punto actual al punto destino h(n). La función f(n) por lo tanto se calcula así:

$$f_{(n)} = g_{(n)} + h_{(n)} \tag{2.2}$$

Donde:

f(n), es la función de evaluación.

g(n), es la función del costo de la ruta hasta el nodo actual.

h(n), es la función heurística que representa el costo estimado desde el nodo actual hasta el destino.

El algoritmo genera dos listas, una lista abierta y una lista cerrada. En la lista cerrada coloca los nodos a medida que van cumpliendo las condiciones de ruta más corta, cuando se llega al nodo final, la lista cerrada representa todos los nodos que forman parte de la ruta óptima. En la lista abierta coloca los nodos que serán evaluados en ese momento.

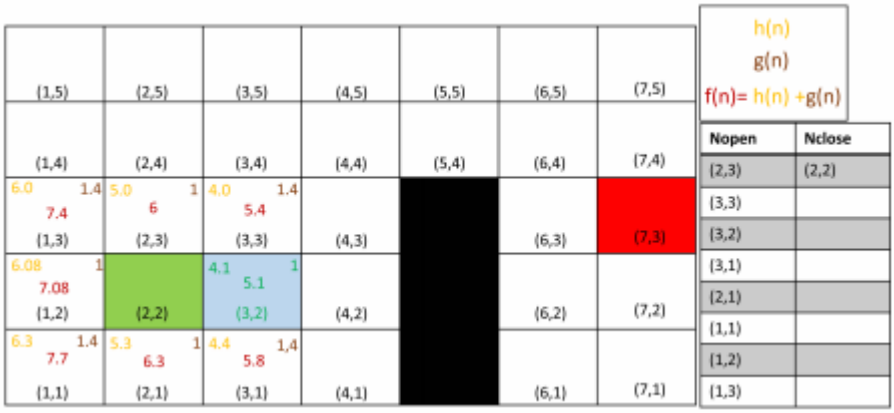


Figura 2.18: Ejemplo A* elección del primer nodo.
(Fuente: Gonzalez E., en [8])

Si vemos el ejemplo de la figura 2.18 el punto de inicio (2,2) está marcado en verde y el destino en rojo (7,3). Por defecto el nodo de inicio (2,2) se ingresa a la lista cerrada ya que a partir de ahí se establecerá la ruta. Para comenzar la búsqueda se evalúan todos los nodos alrededor del nodo de inicio, por lo tanto, estos nodos pasan a formar parte de la lista abierta. Para cada nodo que está en la lista abierta se calcula su función de evaluación f(n), para calcular la función h(n) se suele utilizar la distancia de Manhattan hacia el destino. De todos los nodos evaluados podemos ver que el que tiene la función f(n) con menor valor es el nodo (3,2). Por lo tanto, este nodo pasa a formar parte de la lista cerrada y se retira de la lista abierta.

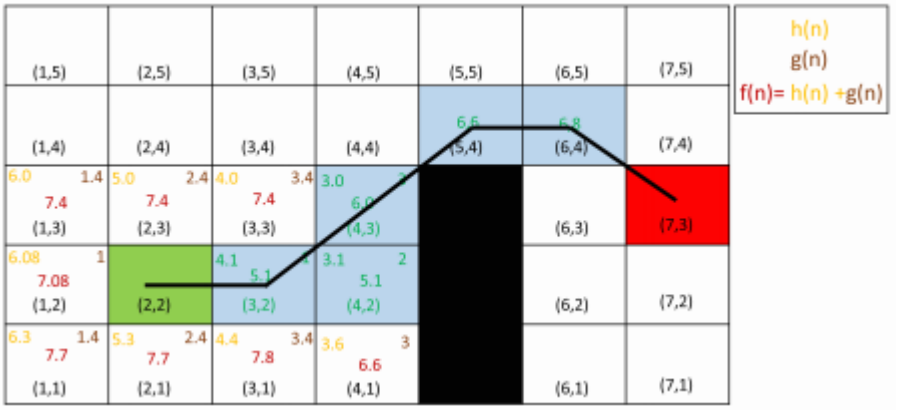


Figura 2.19: Ejemplo A*. Elección de la ruta más corta entre dos puntos. (Fuente: Gonzalez E., en [8])

Utilizando ahora el nodo (3,2) como nodo actual, se procede a evaluar todos los vecinos y así el proceso se repite. Ingresando nodos a la lista abierta, evaluando la función f(n) de cada nodo y el ganador con el valor de la función de evaluación menor va hacia la lista cerrada. La figura 2.19 nos muestra un ejemplo de la ruta optima hallada para este caso.

2.8. Lenguaje de Programación: Python

Python fue creado en 1999 y y20 años después es uno de los lenguajes más populares del mundo. Según el Python Institute en [22], “Python es un lenguaje de programación de alto nivel, interpretado, orientado a objetos y de uso generalizado con semántica dinámica, que se utiliza para la programación de propósito general” (2019).

También se tiene la definición de la Python Software Foundation en [23]:

> Python es un lenguaje de programación interpretado, interactivo y orientado a objetos. Incorpora módulos, excepciones, tipificación dinámica, tipos de datos dinámicos de muy alto nivel y clases. Admite múltiples paradigmas de programación más allá de la programación orientada a objetos, como la programación funcional y de procedimientos. Python combina una potencia notable con una sintaxis muy clara. …También se puede usar como un lenguaje de extensión para aplicaciones que necesitan una interfaz programable. Finalmente, Python es portátil: se ejecuta en muchas variantes de Unix, incluidas Linux y macOS, y en Windows. (2019)

Actualmente hay dos versiones de Python: Python 2 y Python 3. Aunque a simple vista son muy parecidas, las secuencias escritas en Python 2 no pueden ser ejecutadas en Python 3 y viceversa. Python 2 es una versión más tradicional derivada del clásico lenguaje C mientras que Python 3 es un lenguaje completamente diferente.

Para este proyecto se está tomando en consideración que la programabilidad en la red es relativamente nueva, por lo tanto, se utilizará Python 3 para están a la vanguardia, ya que no es necesario mantener compatibilidad hacia atrás con otros programas.

CAPITULO III
DESARROLLO DEL TRABAJO DE TESIS

En el presente capítulo, se explica la metodología que será aplicará para el desarrollo del presente trabajo, así como cada una de las etapas que forman parte de la red flexible que deberán acomodarse para establecer rutas específicas de tráfico cuando un usuario solicita un servicio. También se explicará cómo se validará la hipótesis a través de la simulación de la red con Mininet y el análisis con Octave.

3.1. Metodología de desarrollo

El fin de este proyecto es demostrar que es posible ahorrar energía en un centro de datos apagando los conmutadores que forman parte de su red; sin embargo, como los centros de datos ofrecen una gran variedad de servicios aplicaremos el proyecto a un caso particular. En este proyecto nos enfocaremos en un centro de datos que ofrece un servicio PaaS, para esto aloja máquinas virtuales que pueden ser solicitadas por los usuarios bajo demanda durante un periodo de trabajo específico.

El presente trabajo se compone de 3 bloques diferenciados, según el flujo establecido en la Figura 3.1. Cada bloque abarca funciones específicas y el flujo muestra el orden en el que deben ser desarrollados para completar la totalidad del proyecto, realizar las pruebas y la validación.

Figura 3.1: Bloques del Proyecto.

Dentro de cada bloque además se ha identificado el flujo que se seguirá para el diseño de cada uno de ellos:

Bloque de infraestructura. En este bloque se definen características de la infraestructura como: la topología, los elementos físicos que forman parte de ElasticTree y el modelo energético de la red del centro de datos. Ver figura 3.2.

Figura 3.2: Flujo en el bloque de infraestructura.

Bloque de control. En este bloque se define la lógica de control, que incluye: la detección del origen y destino del flujo, la aplicación de un algoritmo para identificar la mejor ruta, la actualización de las tablas de flujo y el control del on/off de los puertos de los conmutadores. Ver figura 3.3.

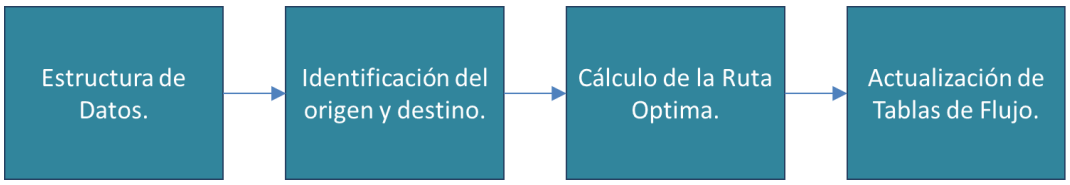


Figura 3.3: Flujo en el bloque de control.

Bloque de Simulación y Resultados. En este bloque se despliegan los entornos de simulación de la red SDN y el cálculo del ahorro de energía en Octave. Ver figura 3.4.

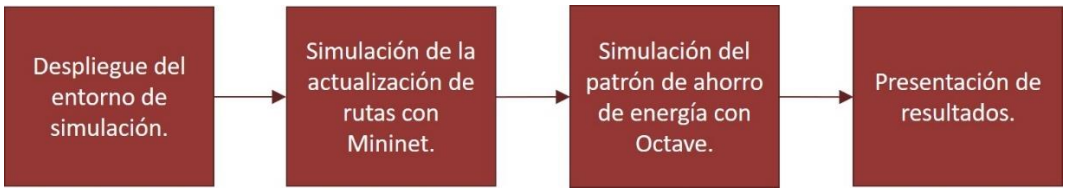


Figura 3.4: Flujo en el bloque de infraestructura.

3.2. Bloque de Infraestructura

En este bloque veremos todo lo referente a la infraestructura de red del centro de datos a nivel de las conexiones físicas y lógicas, se define la topología más adecuada y como se implementará ElasticTree y el modelo energético de red. Comenzamos describiendo de manera general el escenario del proyecto para luego ir detallando los componentes

necesarios y finalizaremos con un escenario particular donde se realizarán las pruebas del estudio.

El Centro de datos ofrecerá PaaS (Plataforma como Servicio), es decir los usuarios tendrán acceso a un servidor virtual donde podrán instalar desde el sistema operativo hasta aplicaciones. Estos servidores serán solicitados bajo demanda por los usuarios durante periodos de trabajo que fluctuarán en función de las necesidades de estos; los servidores pueden ser solicitados en cualquier momento del día.

La gestión de las máquinas virtuales (encendido, apagado, hibernación, transferencia) así como el hardware donde están alojadas, no es parte del alcance de este proyecto y se considerará que están encendidas en todo momento.

La Figura 3.5 nos muestra una vista general del centro de datos, tenemos en un extremo los servidores virtuales y en el otro extremo los usuarios quienes solicitaran el uso de los servidores. En medio esta la red de datos que interconecta ambos extremos; esta red la podemos dividir en dos partes, la red del centro de datos que interconecta los servidores y la red de usuarios que interconecta a los usuarios. Ambas redes se conectan a través de uno o más enlaces.

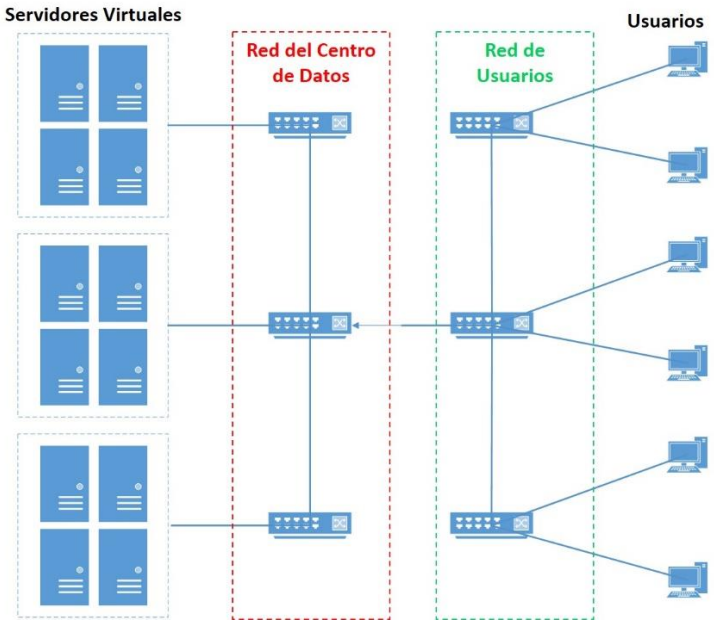


Figura 3.5: Esquema de conexión de un centro de datos

Debido a que queremos reducir el consumo de energía en el centro de datos, será la red del centro de datos la que vamos a analizar; quedando la red de usuarios fuera del

alcance de este proyecto. Por lo tanto, la red de usuarios será representada como un único nodo que se conecta a la red del centro de datos.

En la Figura 3.6 mostramos una topología BasicTree con conmutadores *Top of Rack* para representar la red del centro de datos, utilizamos esta topología temporalmente para poder identificar donde realizaremos la gestión energética de enlaces. Los servidores se conectan a través de conmutadores de acceso y los usuarios se conectan a la red del centro de datos a través de uno o más enlaces desde la red de usuarios (enlaces de color verde). Sin embargo, cuando se requiere establecer una comunicación entre los usuarios y los servidores es necesario atravesar todas las capas de la red del centro de datos (enlaces de color rojo).

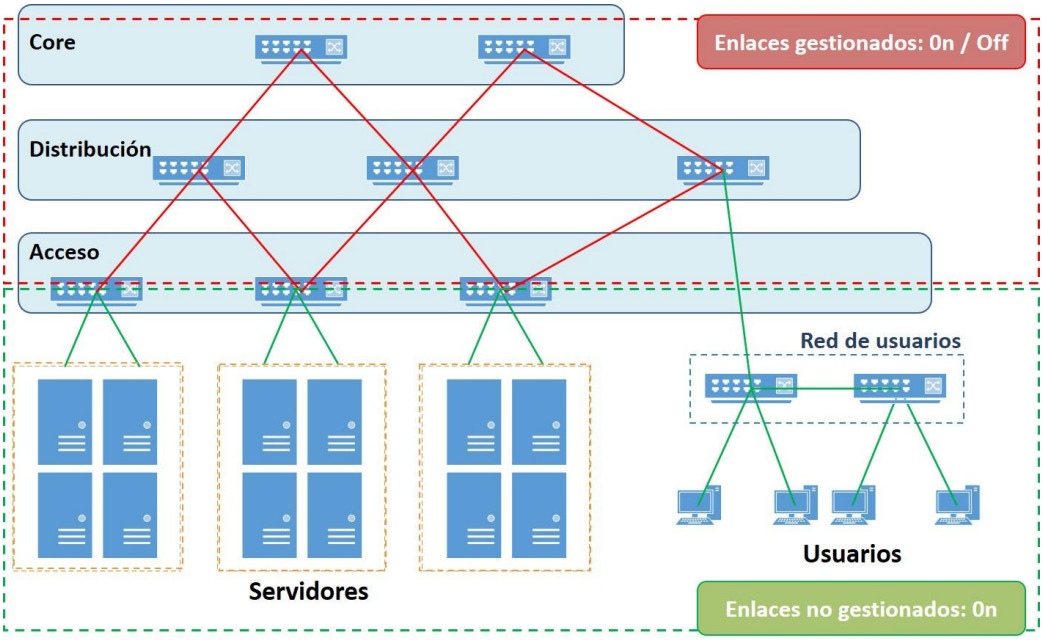


Figura 3.6: Esquema de conexión de un centro de datos

Debido a que se requiere garantizar la conectividad en todo momento, en la red del centro de datos, se mantienen enlaces redundantes activos a la espera de cursar tráfico. Estos, a pesar de que no están siendo utilizados consumen energía, es aquí donde se debe gestionar el encendido/apagado de enlaces para mejorar el rendimiento energético. Estos enlaces se representan de color rojo en la figura 3.6.

Dado que se da por supuesto que los tanto los servidores y los usuarios están siempre encendidos, los enlaces de los servidores hacia los conmutadores de acceso siempre estarán activos y no serán gestionados. La red de usuarios también está fuera del alcance

de la gestión, como se mencionó en un párrafo anterior. Estos enlaces se representan de color verde en la figura 3.6.

3.2.1. Selección de la Topología a utilizar

En la figura anterior se representó la red del centro de datos con una topología BasicTree, sin embargo, tal como se menciona en el marco teórico la necesidad de disponibilidad de comunicación de los servicios, requiere una topología de red confiable, redundante y escalable; una red que garantice que en cualquier momento los servidores pueden ser alcanzados sin correr el riesgo de que los enlaces se encuentren ocupados.

Para conseguir este fin se debe utilizar una red de tipo CLOS: de acuerdo con el marco teórico se recomienda el uso de una topología FatTree, basada en una red CLOS colapsada. Se propone, por lo tanto, para la red del centro de datos una topología FatTree con $n = 4$.

Utilizando las fórmulas de Fat-tree con $n = 4$, tenemos:

Conmutadores en el core.

$$\left(\frac{n}{2}\right)^2 = \left(\frac{4}{2}\right)^2 = 4 \quad (3.1)$$

Servidores.

$$\left(\frac{n^3}{4}\right) = \left(\frac{4^3}{4}\right) = 16 \quad (3.2)$$

Un pod es la agrupación de $\left(\frac{n}{2}\right) = 2$ conmutadores de acceso y $\left(\frac{n}{2}\right) = 2$ conmutadores de la capa de distribución, en total 4 conmutadores por cada pod. Adicionalmente cada conmutador de acceso tiene 2 enlaces disponibles para conectar servidores, como hay 2 conmutadores acceso en el pod, entonces tenemos 4 servidores por pod. Y en la red Fat-tree hay n=4 pods.

De lo anterior tenemos 4 pods y en cada pod 4 conmutadores, 2 de acceso y 2 de distribución. El número de enlaces entre capas:

$$\left(\frac{n^3}{4}\right) = \left(\frac{4^3}{4}\right) = 16 \quad (3.3)$$

Para finalizar, en cada capa, cada conmutador utiliza $\left(\frac{n}{2}\right) = 2$ enlaces para conectarse hacia el sur y $\left(\frac{n}{2}\right) = 2$ enlaces hacia el norte, hasta llegar a la capa de core que solo tiene enlaces hacia el sur.

En resumen, podemos tener para una topología FatTree con $n = 4$:

Número total de conmutadores:

$$numSWTotal = \left(\frac{5}{4}\right) \times n^2 = \left(\frac{5}{4}\right) \times 4^2 = 20 \quad (3.4)$$

Número de conmutadores de acceso:

$$numSWTotal = \left(\frac{5}{4}\right) \times n^2 = \left(\frac{5}{4}\right) \times 4^2 = 20 \quad (3.5)$$

Número de conmutadores de distribución:

$$numSWDist = \left(\frac{n^2}{2}\right) = \left(\frac{4^2}{2}\right) = 8 \quad (3.6)$$

Número de conmutadores de core:

$$numSWCore = \left(\frac{n^2}{4}\right) = \left(\frac{4^2}{4}\right) = 8 \quad (3.7)$$

Número de servidores:

$$num\ Servidores = \left(\frac{n^3}{4}\right) = \left(\frac{4^3}{4}\right) = 16 \quad (3.8)$$

Número de puertos conectados:

$$numPuertos = \left(\frac{5}{4}\right) \times n^3 = \left(\frac{5}{4}\right) \times 4^3 = 80 \quad (3.9)$$

Número de enlaces:

$$numEnlaces = \left(\frac{n^3}{4}\right) \times 3 = \left(\frac{4^3}{4}\right) \times 3 = 48 \quad (3.10)$$

En la figura 3.7 podemos observar cómo quedaría la topología física del centro de datos. Los enlaces que entraran al cálculo energético se representan de color rojo y de color verde los enlaces que estarán siempre energizados.

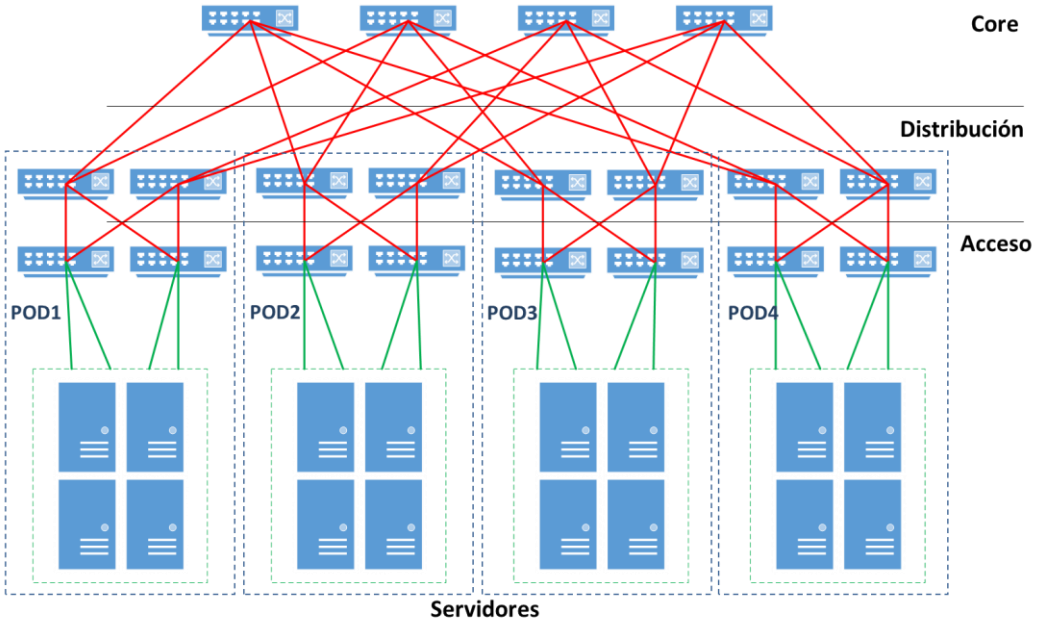


Figura 3.7: Red del centro de datos, topología FatTree con $n = 4$

Para este proyecto asumiremos que la fuente de tráfico de datos que se dirige hacia los servidores proviene desde la red de usuarios. Por lo tanto, el tráfico fluirá *inter-pods*, y no se considerará tráfico *intra-pods*.

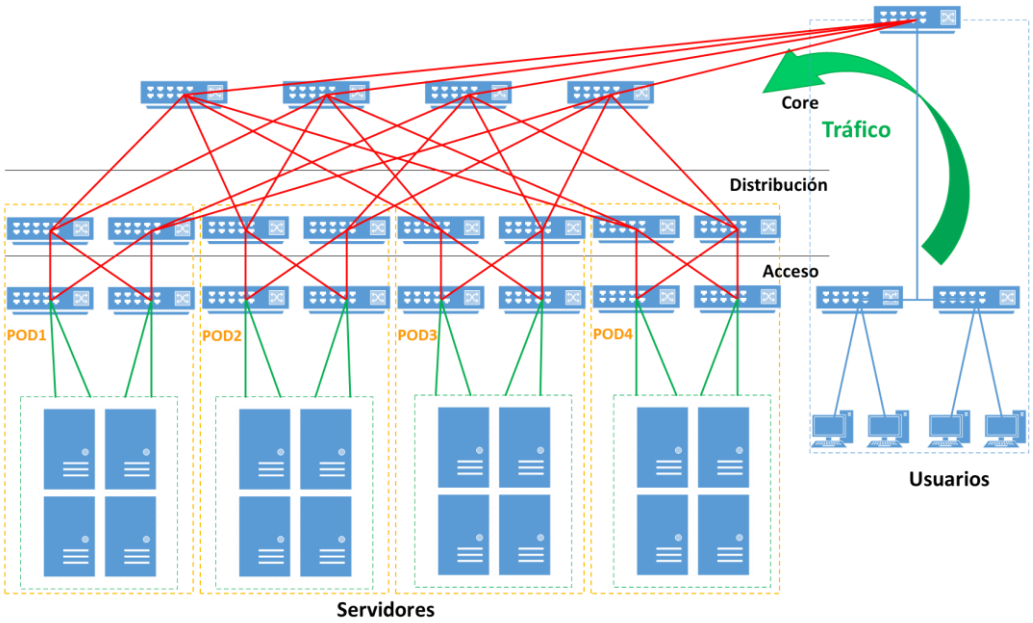


Figura 3.8: Red del centro de datos y red de usuarios.

Como ya se ha mencionado anteriormente la red de usuarios no será considerada en el cálculo energético, por lo tanto, será tomada solo como una fuente de tráfico que se conecta al core del centro de datos, como podemos observar en la figura 3.8. Sin embargo,

para esto como se observa en la figura anterior es necesario que los conmutadores de core habiliten un puerto adicional cada uno para cursar este tráfico. Por lo tanto, al número de puertos conectados totales hay que agregarle un puerto por cada conmutador de core. De las fórmulas anteriores tenemos:

$$numPuertosTotal = \left(\frac{5}{4}\right) \times n^3 + numSWCore = \left(\frac{5}{4}\right) \times 4^3 + 4 = 84 \qquad (3.11)$$

Para hallar el número de enlaces totales, debemos agregarle los 4 enlaces de los conmutadores de core hacia el conmutador monitor:

$$numEnlacesTotal = \left(\frac{n^3}{4}\right) \times 3 + numSWCore = \left(\frac{4^3}{4}\right) \times 3 + 4 = 52 \qquad (3.12)$$

En la tabla 3.1 enumeramos los elementos que participan en la topología del centro de datos. A partir de esta topología generaremos una matriz de nodos y enlaces que utilizaremos para el cálculo de la mejor ruta.

Tabla 3.1: Elementos de la topología

Elemento	Cantidad
Conmutadores en el Core	4
Conmutadores en la Distribución	8
Conmutadores en el Acceso	8
Servidores	16
Enlaces servidor - acceso	16
Enlaces acceso - distribución	16
Enlaces distribución - core	16
Puertos conectados	84

3.2.2. Componentes de ElasticTree y SDN

En el punto anterior se ha seleccionado una topología FatTree con $n = 4$, si bien es cierto, esta topología garantiza redundancia y disponibilidad; el costo de estos beneficios es el alto consumo energético que produce mantener todos los enlaces y equipos funcionando. Por lo tanto, en el 2010 Heller et al. en [12] nos proponen ElasticTree, una alternativa que ajusta de forma dinámica el consumo de energía de la red de un centro de datos en función del tráfico que la atraviesa.

Aplicaremos el enfoque de ElasticTree a partir de la topología FatTree seleccionada, de tal manera que, en un momento dado, solo un subconjunto de enlaces FatTree estará encendido. Se encenderán solo los enlaces que están cursando tráfico en ese momento. ElasticTree está formado por 3 componentes funcionales, tal como se ha detallado en el marco teórico: optimizador, enrutador y control de energía. Estos tres componentes funcionales estarán implementados a través de scripts en el controlador SDN.

En este punto introducimos un elemento más a la topología presentada en los puntos anteriores, el servidor que cumplirá el rol de controlador SDN. El controlador SDN será tratado en detalle más adelante en el bloque de control, aquí lo veremos como un elemento adicional de hardware que deberemos incluir en nuestra topología de acuerdo con la arquitectura SDN. El controlador SDN utilizará como protocolo para la interface de puente sur OpenFlow1.3 y este deberá comunicarse con cada conmutador Openflow que participe de la red SDN. Por lo tanto, todos los conmutadores que participen del cálculo energético deben formar parte de la red SDN.

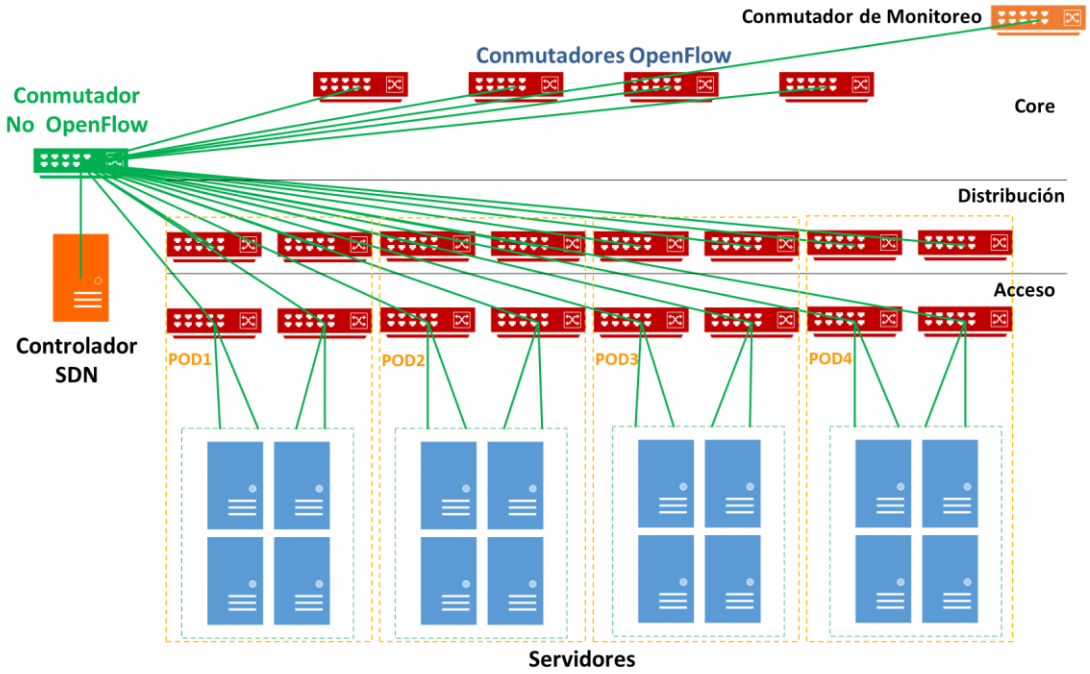


Figura 3.9: Red de control OpenFlow.

Desde el punto de vista del control SDN la red del centro de datos se verá como en la figura 3.9, aquí podemos observar el controlador conectado con cada conmutador Openflow que será incluido en calculo energético. El control será realizado fuera de banda, es decir, que el tráfico OpenFlow irá del controlador SDN a los conmutadores OpenFlow por una red separada de la red que cursará el tráfico de datos en usuarios y servidores. En

la figura se observa la red de control con enlaces de color verde, un conmutador conecta el controlador SDN con los conmutadores OpenFlow, este conmutador no necesita ser OpenFlow ya que su función se centra en distribuir el tráfico de control. La red de control no entrará de manera dinámica al cálculo de control energético ya que estos enlaces deben estar siempre encendidos intercambiando tráfico OpenFlow.

Para que el controlador SDN pueda tomar las decisiones necesarias para el control de tráfico y de energía se requiere recibir la información del tráfico que cruzará a través de la red del centro de datos. Para esto debemos identificar un conmutador que actuara como hardware de monitoreo de tráfico. Seleccionaremos como conmutador monitor al conmutador que recibe el tráfico de entrada proveniente de la red de hosts, la información con los parámetros de los paquetes de entrada serán enviados al controlador SDN. El detalle del procesamiento de estos paquetes se estudia en el bloque de control.

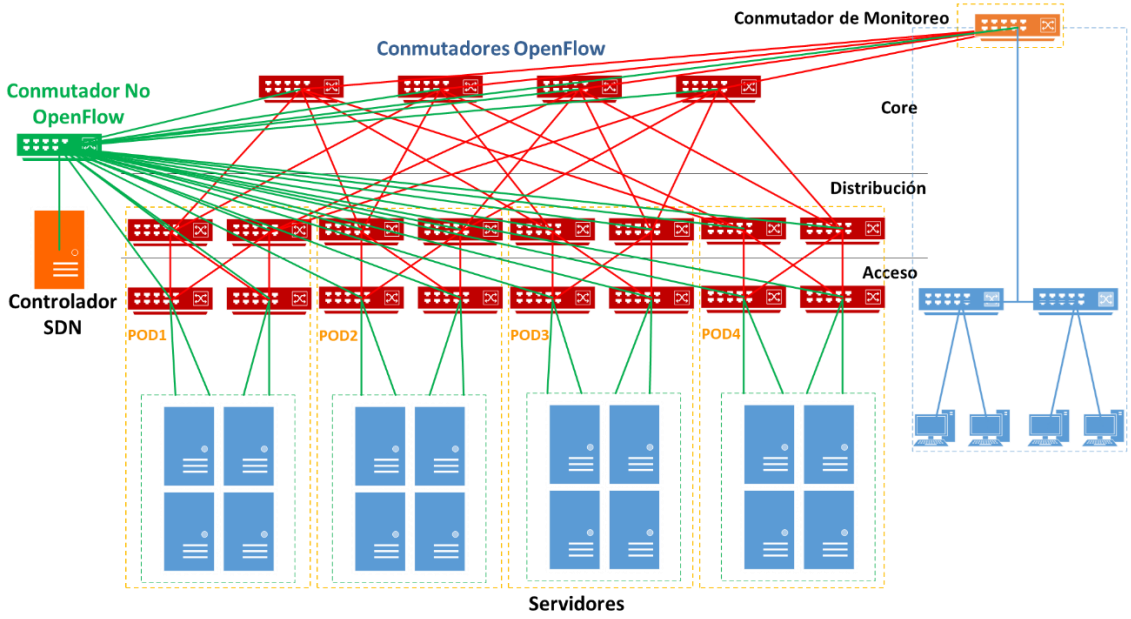


Figura 3.10: Topología de red final del centro de datos.

Agregando la red de control al centro de datos obtenemos la figura 3.10, donde observamos la topología final. Los conmutadores y enlaces en color rojo participan del cálculo energético y cursan tráfico de datos entre los usuarios y servidores. Los conmutadores y enlaces en color verde no participan del cálculo energético por lo que permanecerán siempre encendidos, y cursarán el tráfico de control OpenFlow. El conmutador de monitoreo en color anaranjado forma parte de la red de control, pero no ingresa al cálculo energético. La red de usuarios tampoco participa del cálculo de energía y es la fuente del tráfico hacia los servidores que es recibido por el conmutador de monitoreo formada por enlaces de color azul.

3.2.3. Componentes del modelo energético

Para el modelo energético utilizaremos la fórmula que nos propone el artículo de Mahadevan et al en [15], según lo expuesto en el marco teórico; por lo tanto, por cada conmutador la fórmula es la siguiente:

$$P_{switch} = P_{chasis} + n_{linecards} \times P_{linecard} + \sum_{i} nport_{ri} \times P_{ri} \times F_u \quad (3.13)$$

- P_{switch} -> Energía consumida por el conmutador.
- P_{chasis} -> Energía consumida por el chasis del conmutador.
- $n_{linecards}$ -> Número de tarjetas de línea.
- $P_{linecard}$ -> Energía consumida por una tarjeta de línea sin considerar el consumo por puerto.
- P_{ri} -> Energía consumida por puerto activo.
- $nport_{ri}$ - > número de puertos activos de tipo i.
- F_u -> Es un factor de escala que se utiliza por puerto. Se considerará $F_u = 1$ para todos los puertos.

A partir de la formula anterior propondremos la fórmula de cálculo energético por conmutador para el proyecto tomando en cuenta las siguientes consideraciones:

- Se considera una red Fat-tree n = 4.
- Todos los conmutadores son del mismo tipo y tienen el mismo número de puertos. Se consideran 4 puertos para conectarse al Fat-tree dos hacia el norte 2 hacia el sur. Se considera un puerto adicional para la red de control SDN. Todos los puertos serán GigabitEthernet (1Gbps).
- Por lo anterior, se considera que la potencia de la tarjeta de línea ($P_{linecard}$) y la de potencia del chasis (P_{chasis}) será constante, por lo tanto, para fines prácticos ambas potencias las consideraremos únicamente como: P_{chasis}
- Como todos los enlaces serán de 1Gbps entonces i = 1.
- Cada flujo de tráfico consume como máximo 0.2 Gbps.

Para obtener la potencia que consumen los conmutadores y sus puertos nos basaremos en los resultados del trabajo de Mahadevan P. et al en [15]. Una conclusión importante de su trabajo es: “el consumo real de potencia de los componentes de un conmutador varia ampliamente del valor de consumo nominal que podemos ver en las

fichas técnicas". Por lo tanto, en su trabajo a través de pruebas experimentales, con diferentes marcas, elaboró unas tablas de consumo promedio de potencia. Para este proyecto se ha decidido utilizar estas tablas para determinar el consumo de potencia de los componentes de un conmutador, ya que se acercan más a un caso real. A partir de la tabla 2.3 del marco teórico, seleccionamos los valores del conmutador del modelo A, considerando que es un conmutador con 48 puertos se asignan los valores a los diferentes componentes del conmutador obteniendo los resultados en la tabla 3.2.

Tabla 3.2: Consumo de energía de un conmutador de 48 puertos.

Puertos Activos	Puertos con Tráfico	Consumo de potencia (W)	Componente energizado
Ninguno	Ninguno	151	Chasis (P_{chasis})
Todos	Ninguno	184	48 puerto conectados
Todos	1 Gbps	195	48 puertos cursando tráfico

Con los resultados anteriores obtenemos la potencia que consume el chasis y la potencia por puerto.

P_{chasis} = 151 W

$P_{conected}$ = 184 / 48 = 3.83 W

P_r = 195 / 48 = 4.0 W

Para los conmutadores de acceso utilizaremos la siguiente fórmula para calcular la potencia:

$$Psa_k = P_{chasis} + nports \times P_r \quad (3.14)$$

$$Psa = \sum_k Psa_k \quad (3.15)$$

$$Psa = \sum_k (P_{chasis} + nports_k \times P_r) \quad (3.16)$$

$$Psa = nsa \times (P_{chasis}) + \sum_k (nports_k \times P_r) \quad (3.17)$$

Donde:

Psa_k, es la potencia del *k-ésimo* conmutador de acceso.

Psa, es la potencia de todos los conmutadores de acceso.

$nsa > 0 \rightarrow 8$, número de conmutadores encendidos

$nports_k => 0 \rightarrow n$, numero de puertos encendidos.

$k => 1 \rightarrow nsa$

Para los conmutadores de distribución utilizaremos la siguiente fórmula para calcular la potencia:

$$Psd_k = P_{chasis} + nports \times P_r \quad (3.18)$$

$$Psd = \sum_k Psd_k \quad (3.19)$$

$$Psd = \sum_k (P_{chasis} + nports_k \times P_r) \quad (3.20)$$

$$Psd = nsd \times (P_{chasis}) + \sum_k (nports_k \times P_r) \quad (3.21)$$

Donde:

Psd_k, es la potencia del *k-ésimo* conmutador de distribución.

Psd, es la potencia de todos los conmutadores de distribución.

$nsd => 0 \rightarrow 8$, número de conmutadores encendidos.

$nports_k => 0 \rightarrow n$, numero de puertos encendidos.

$k => 1 \rightarrow nsd$

Para los conmutadores de Core utilizaremos la siguiente fórmula para calcular la potencia:

$$Psc_k = P_{chasis} + nports \times P_r \quad (3.22)$$

$$Psc = \sum_k Psc_l \quad (3.23)$$

$$Psc = \sum_k (P_{chasis} + nports_k \times P_r) \quad (3.24)$$

$$Psc = nsc \times (P_{chasis}) + \sum_k (nports_k \times P_r) \quad (3.25)$$

Donde:

Psc_l, es la potencia del *k-ésimo* conmutador de distribución.

Psc, es la potencia de todos los conmutadores de distribución.

$nsc => 0 \rightarrow 4$ número de conmutadores encendidos.

$nports_k => 0 \rightarrow (n + 1)$

$k => 1 \rightarrow nsc$

La fórmula final para el cálculo de potencia total es:

$$Pst = Psa + Psd + Psc \quad (3.26)$$

$$Pst = (nsa + nsd + nsc) \times P_{chasis} + \sum_{k} (nports_k \times P_r) \quad (3.27)$$

$$Pst = nst \times P_{chasis} + \sum_{k} (nports_k \times P_r) \quad (3.28)$$

Donde:

Pst, es la potencia de todos los conmutadores de la red.

$nst = nsa + nsd + nsc$

$nports_k => 0 \rightarrow (n + 1)$

$k => 1 \rightarrow nst$

Es necesario aclarar que no se considerará al controlador como parte del modelo de consumo de potencia. Esto se debe a que el controlador se ejecuta en una máquina virtual, que podría ejecutarse en el hardware que ya forma parte del centro de datos. Por lo que el cálculo del consumo de energía quedará fuera del alcance de este proyecto.

3.3. Bloque de Control

Una vez que tenemos la topología establecida y ya hemos definido el modelo para el cálculo energético, determinaremos como el controlador realizará el proceso de selección de ruta optima y como seleccionará que conmutadores-puertos se mantendrán prendidos o pagados (On/Off) durante un periodo de tiempo.

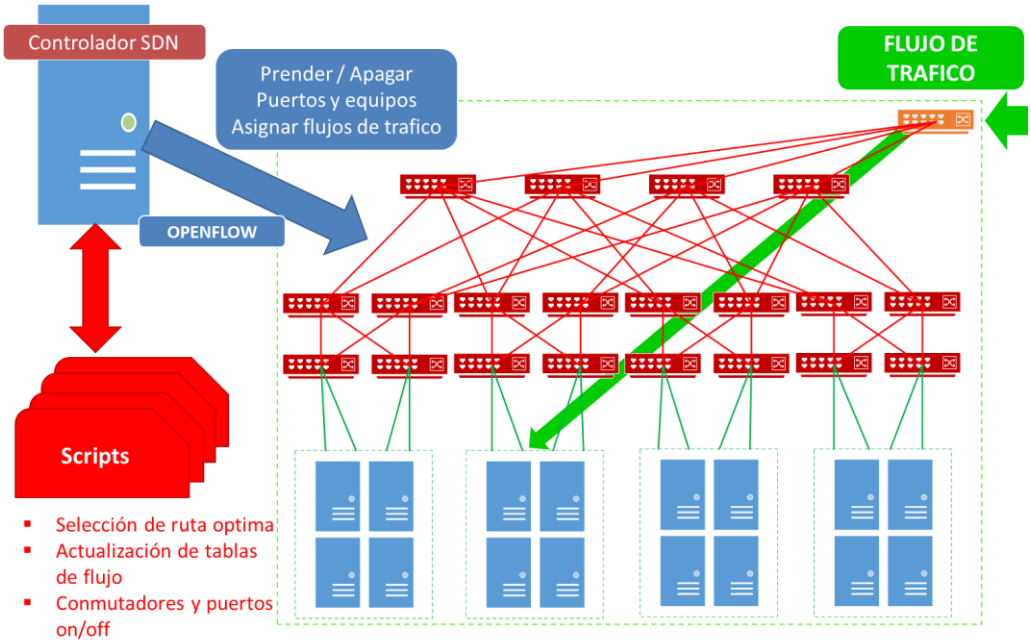


Figura 3.11: Despliegue del controlador

Partiremos del gráfico de la figura 3.11 donde observamos como el controlador interactúa con las diferentes partes de la red, el flujo de tráfico ingresa por el conmutador de monitoreo quien enviará al controlador la solicitud de acceso. El controlador a través de scripts ejecuta un algoritmo de optimización y obtiene la ruta optima, con este resultado utiliza OpenFlow para actualizar las tablas de flujo de los conmutadores, a su vez establece que enlaces se deben apagar y encender. A partir de ahí realizaremos el cálculo energético.

El bloque de control valga la redundancia, está a cargo del controlador ahí se ejecutarán los scripts necesarios que permitirán establecer el conjunto mínimo de dispositivos disponibles necesarios para operar en un momento dado. En la figura 3.12 hemos reconocido los componentes funcionales que realizaran la tarea de control, estos son los componentes de ElasticTree que hemos adaptado a este caso particular.

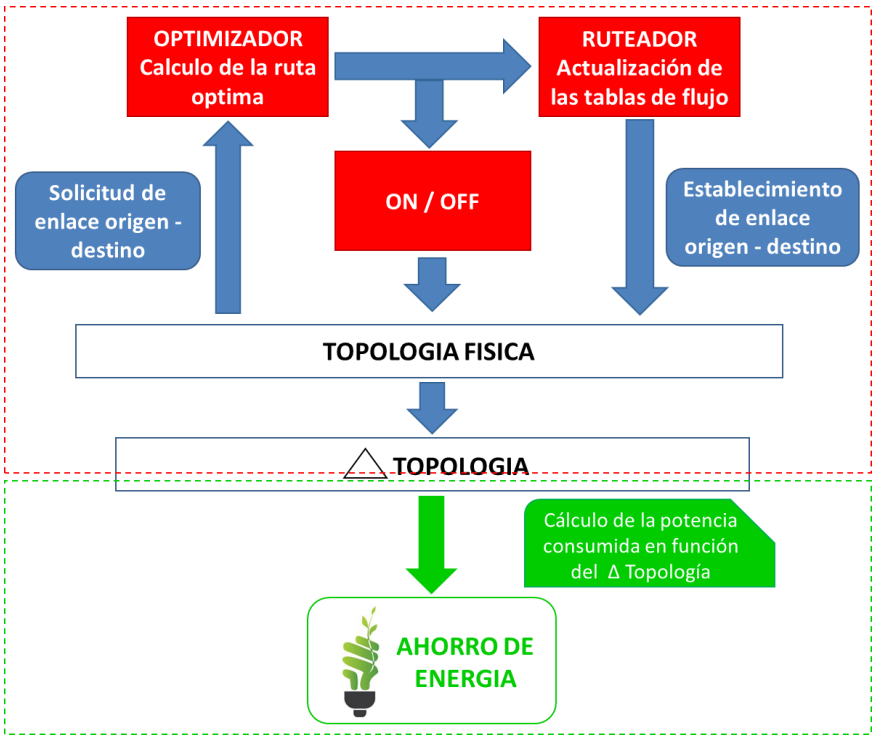


Figura 3.12: Diagrama funcional del bloque de control

- El optimizador, calcula la ruta optima en función de la solicitud de tráfico origen destino.
- El ruteador actualiza las tablas de flujo de los conmutadores.
- El bloque on-off, enciende y/o apaga los enlaces y conmutadores según el resultado del optimizador.

Estos tres bloques actúan sobre la topología del centro de datos; como resultado de estos cálculos obtenemos un ΔTopología, es decir un sub-conjundo de la topología física que representa los enlaces y conmutadores que quedan encendidos después del cálculo. A partir de este ΔTopología se realizan los cálculos energéticos, utilizando Octave.

3.3.1. Estructura de Datos

El control de la red se implementará a través de software utilizando como lenguaje de programación Python. Para que el software pueda ejecutar el algoritmo de optimización y actualizar las tablas de flujo de los conmutadores, es necesario, identificar los recursos que forman parte de la red FatTree. Como ya tenemos definida la topología física y se han definido los componentes que forman parte del cálculo energético, procedemos a identificarlos de forma numérica. Esto es para facilitar los cálculos y la definición de las estructuras de datos que crearemos para trabajar con ellos.

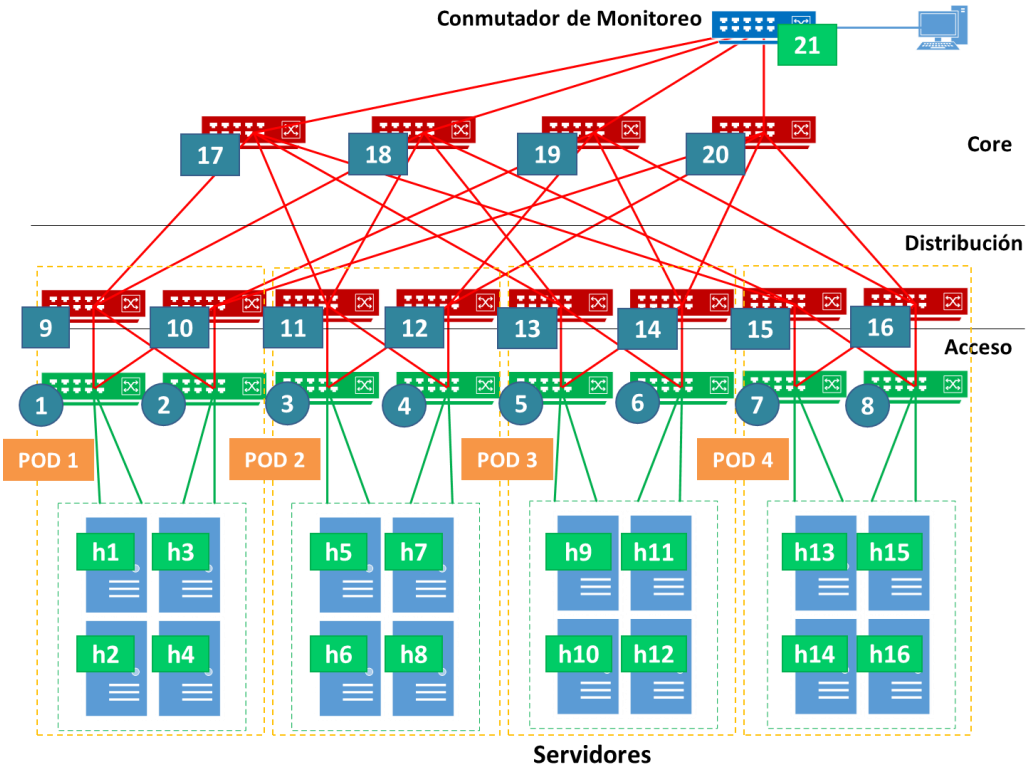


Figura 3.13: Identificación de los componentes de FatTree.

En la Figura 3.13 podemos observar que los servidores se numeran desde h01 hasta h16, los conmutadores de acceso van desde 1 a 8, los conmutadores de distribución de 9 a 16 y los conmutadores de Core de 17 a 20. Adicionalmente se ha numerado el

conmutador de monitoreo como 21, con fines de establecer luego adyacencias, pero este conmutador no participa del cálculo energético.

Como sabemos los conmutadores de acceso estarán siempre encendidos y son los que van del 1 al 8. Es necesario que se tenga esto en cuenta al momento de crear las estructuras de datos. También se debe tener en cuenta que a pesar de que el conmutador de monitoreo no se cuenta en el cálculo energético, los puertos de los conmutadores de Core que van hacia el mismo si están considerados.

Para identificar los puertos asignaremos números de puerto a cada conmutador de acuerdo con la Figura 3.14. De esta manera será posible identificar un puerto específico con una tupla conmutador, puerto. En el caso de los conmutadores de acceso podemos observar que los puertos 1 y 2 están siempre encendidos porque son los que van hacia los servidores; mientras que los puertos 3 y 4 van hacia los conmutadores de distribución. Para los conmutadores de distribución los puertos 1 y 2 van hacia los conmutadores de acceso y los puertos 3 y 4 hacia los conmutadores de core. Y finalmente, los conmutadores de core tienen los puertos 1 al 4 conectados a los conmutadores de distribución y el puerto 5 hacia el conmutador de monitoreo.

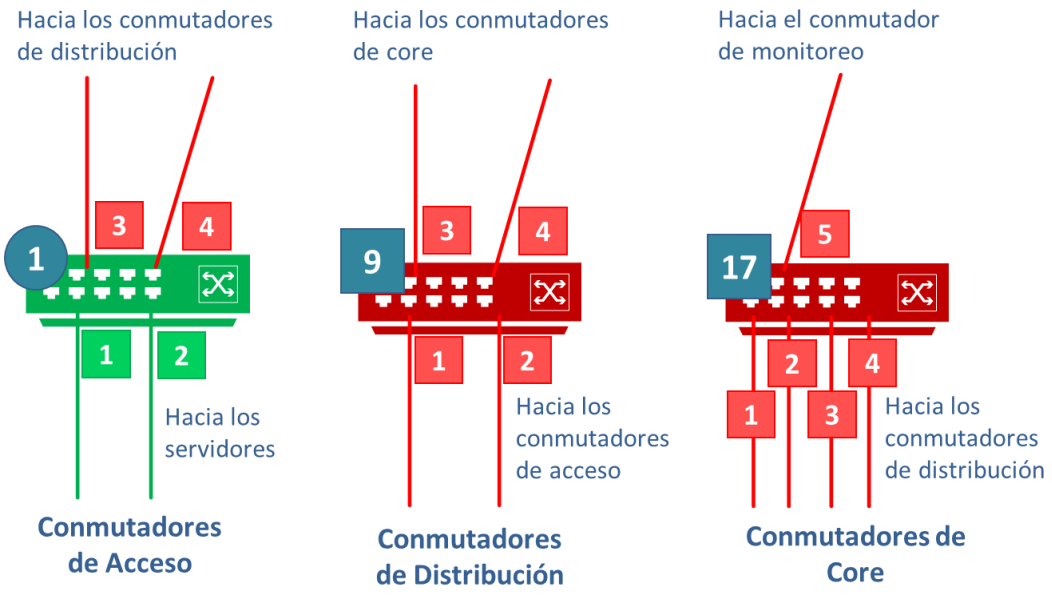


Figura 3.14: Identificación de puertos en los conmutadores.

Establecida la numeración de conmutadores y puertos, ahora se crearán las estructuras de datos que serán utilizadas en el programa de monitoreo y optimización. Las estructuras de datos son matrices que contienen la información de: conmutadores, puertos,

enlaces, estado, adyacencias, cambios de topología, actualizaciones de tablas de flujo y otros. Algunas de estas estructuras se cargan al programa desde archivos csv y otras se crean dinámicamente en memoria.

Las estructuras de datos utilizadas en el proyecto son las siguientes:

1. Matriz de Adyacencias (matAdyacencias): como vemos en la figura 3.15, esta matriz nos permite mapear que dispositivo está conectado físicamente con otro. Se almacena como archivo csv y luego es cargada en memoria por el programa de control. Es una matriz de 21x21, las filas y columnas representan los 21 dispositivos de red dentro de la topología FatTree.

   Como observamos en la Figura 3.15, utilizando los índices (fila,columna) podemos identificar si hay un enlace entre conmutadores.
   Por ejemplo: matAdyacencias(1,9) = 1, lo que significa que entre el conmutador 1 y el 9 hay un enlace físico.

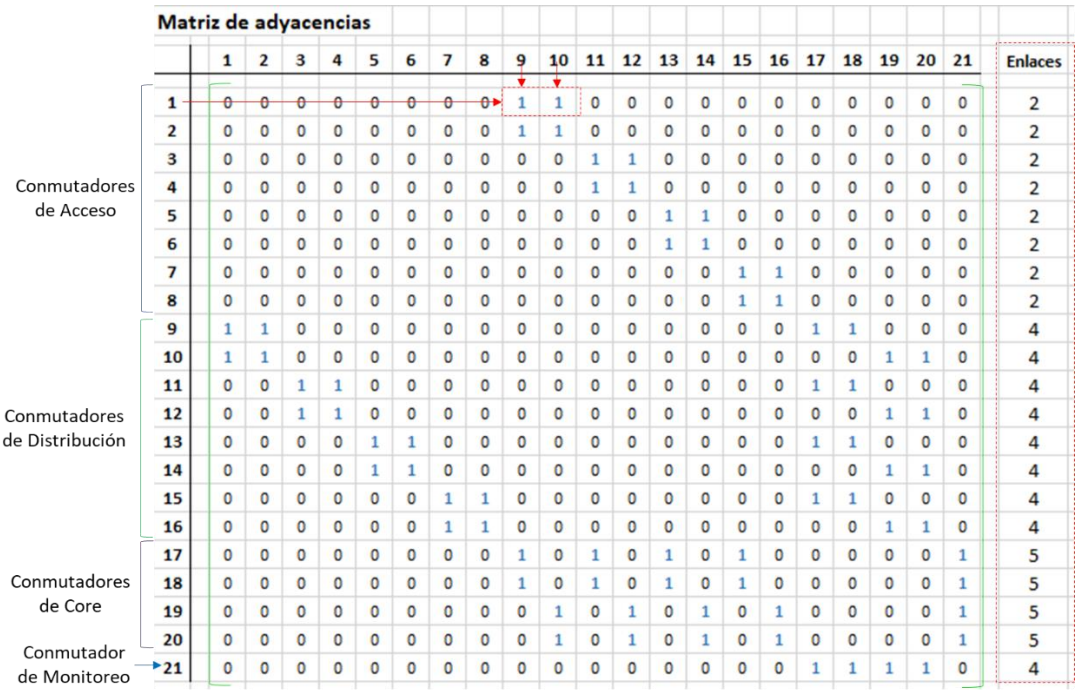


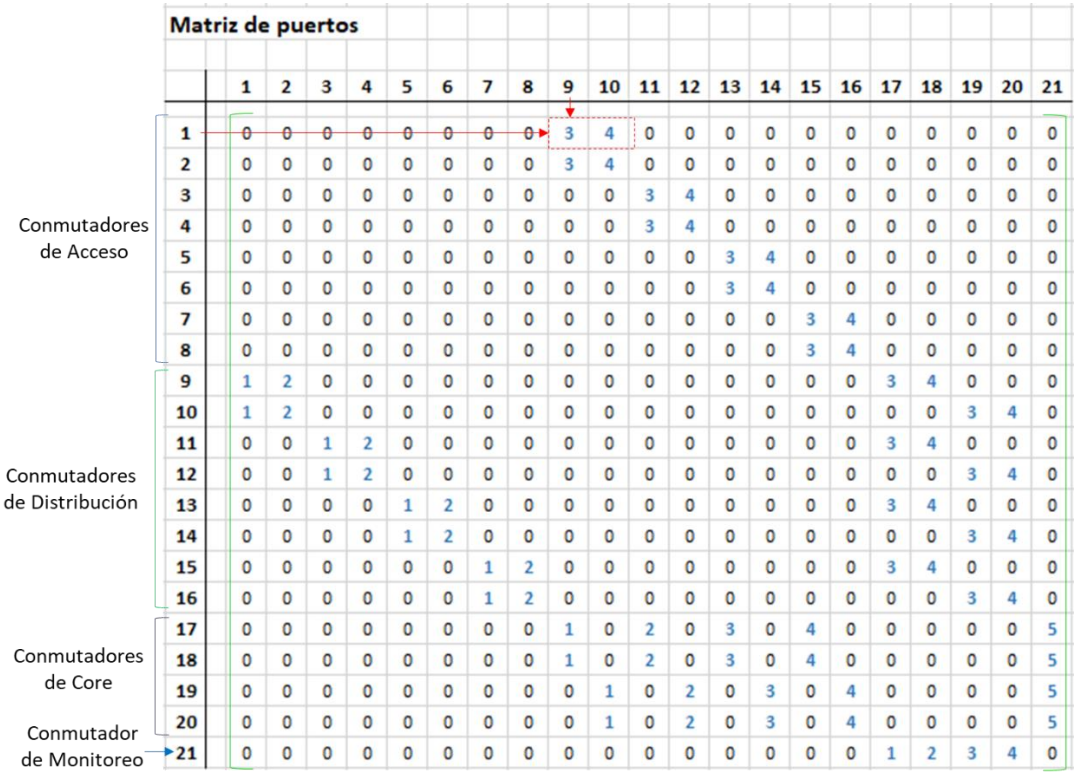
Figura 3.15: Matriz de Adyacencias.

También podemos utilizar esta matriz para contabilizar con cuantos enlaces cuenta cada dispositivo.

2. Matriz de Puertos (matPuertos): como vemos en la figura 3.16 esta matriz nos permite identificar a través de que puertos están conectados dos dispositivos. Es también una matriz 21x21, con las filas y columnas haciendo referencia a los 21 dispositivos de la topología FatTree. Similar a la matriz de adyacencias cada par (fila, columna) nos indica el número del puerto en el conmutador en la fila lo conecta con el conmutador indicado en la columna.

   Ejemplo: matPuertos(1,9) = 3; es decir: el conmutador 1 se conecta a través del puerto 3 al conmutador 9, ver figura 3.16.

   De esta matriz, además podemos obtener los puertos que intervienen en un enlace especifico de la siguiente manera: matPuertos(1,9) = 3 y matPuertos(9,1) = 1, es decir el enlace entre el conmutador 1 y el conmutador 9 es a través de los puertos 3 y 1 respectivamente.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	0	0	0	0	0	0	0	0	3	4	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	3	4	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	3	4	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	3	4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	3	4	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	3	4	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	4	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	4	0	0	0	0	0
9	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	4	0	0	0
10	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	4	0
11	0	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0	3	4	0	0	0
12	0	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	4	0
13	0	0	0	0	1	2	0	0	0	0	0	0	0	0	0	0	3	4	0	0	0
14	0	0	0	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0	3	4	0
15	0	0	0	0	0	0	1	2	0	0	0	0	0	0	0	0	3	4	0	0	0
16	0	0	0	0	0	0	1	2	0	0	0	0	0	0	0	0	0	0	3	4	0
17	0	0	0	0	0	0	0	0	1	0	2	0	3	0	4	0	0	0	0	0	5
18	0	0	0	0	0	0	0	0	1	0	2	0	3	0	4	0	0	0	0	0	5
19	0	0	0	0	0	0	0	0	0	1	0	2	0	3	0	4	0	0	0	0	5
20	0	0	0	0	0	0	0	0	0	1	0	2	0	3	0	4	0	0	0	0	5
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2	3	4	0

Figura 3.16: Matriz de puertos.

3. Matriz de Tráfico (matTrafico): como vemos en la figura 3.17 esta matriz nos permite identificar si los enlaces físicos entre dispositivos están activos o no y cuanto tráfico están cursando. Es también una matriz 21x21, con las filas y columnas haciendo referencia a los 21 dispositivos de la topología FatTree.

Similar a las matrices anteriores cada par (fila, columna) nos indica si el enlace que une a dos conmutadores esta activo y cuando tráfico cursa en un momento dado. El tráfico cursado se expresará en un rango de 0 a 10, donde 0 es inactivo y 10 es congestionado.

Ejemplos, utilizando la figura 3.17:

- matTráfico(1,9) = 1; es decir: el enlace entre los conmutadores 1 y 9, está activo y está cursando un 10% del tráfico que soporta el enlace.
- matTrafico(17,21) = 5, es decir el enlace entre los conmutadores 17 y 21 está activo y cursando un 50% del tráfico soportado por el enlace.

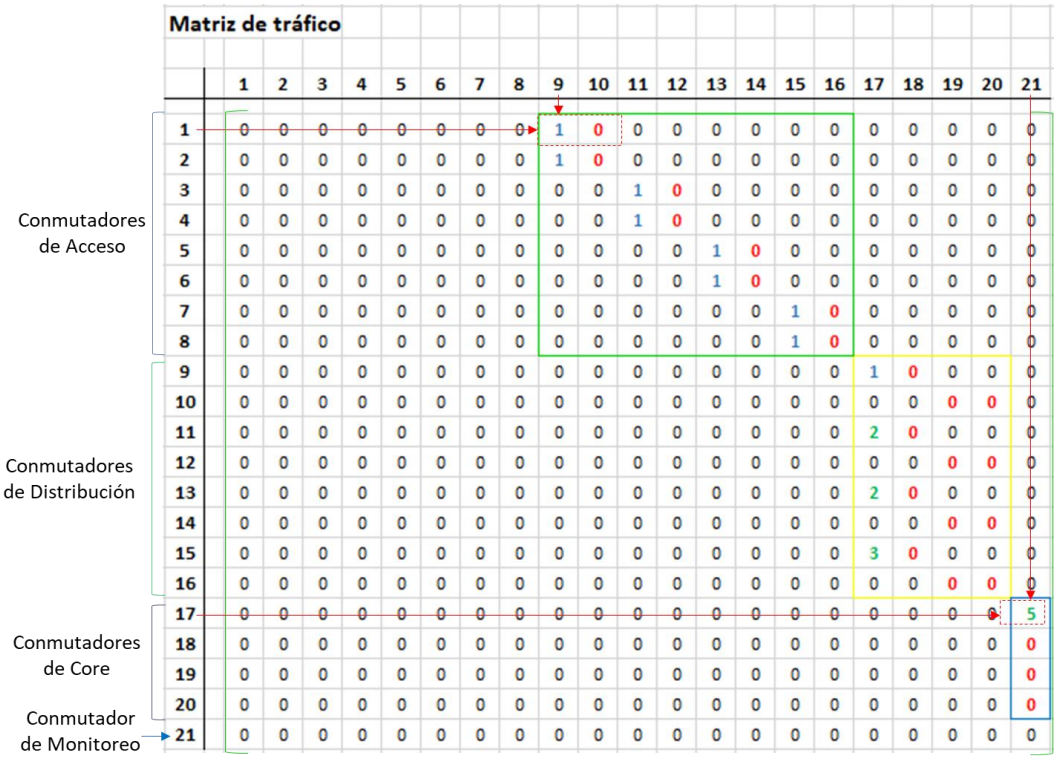


Figura 3.17: Matriz de tráfico.

4. Matriz de Cambios en la Topología (matCambiosTopo): como vemos en la figura 3.18, esta matriz almacena los cambios en la topología lógica de la red ElasticTree como un historial; cada vez que encendemos o apagamos un conmutador y/o puerto registramos el cambio en esta matriz. Esta matriz es de 20xn, las 20 filas representan los 20 conmutadores que participan del cálculo energético y las n columnas representan en pares si el conmutador esta activo y cuantos puertos activos tiene. Esta matriz tendrá tantas columnas como registros de cambio almacenados, n debe ser un número par.

En la Figura 3.18 podemos ver un ejemplo de esta matriz con 3 registros, por lo que matCambiosTopo es de dimensiones 20x6. Ahí se observa en las columnas 3 y 4 que conmutadores están activos y cuantos puertos activos tienen. Por ejemplo matCambiosTopo(9,3) = 1, quiere decir que el conmutador 9 está activo y matCambiosTopo(9,4) = 2, quiere decir, que tiene 2 puertos activos.

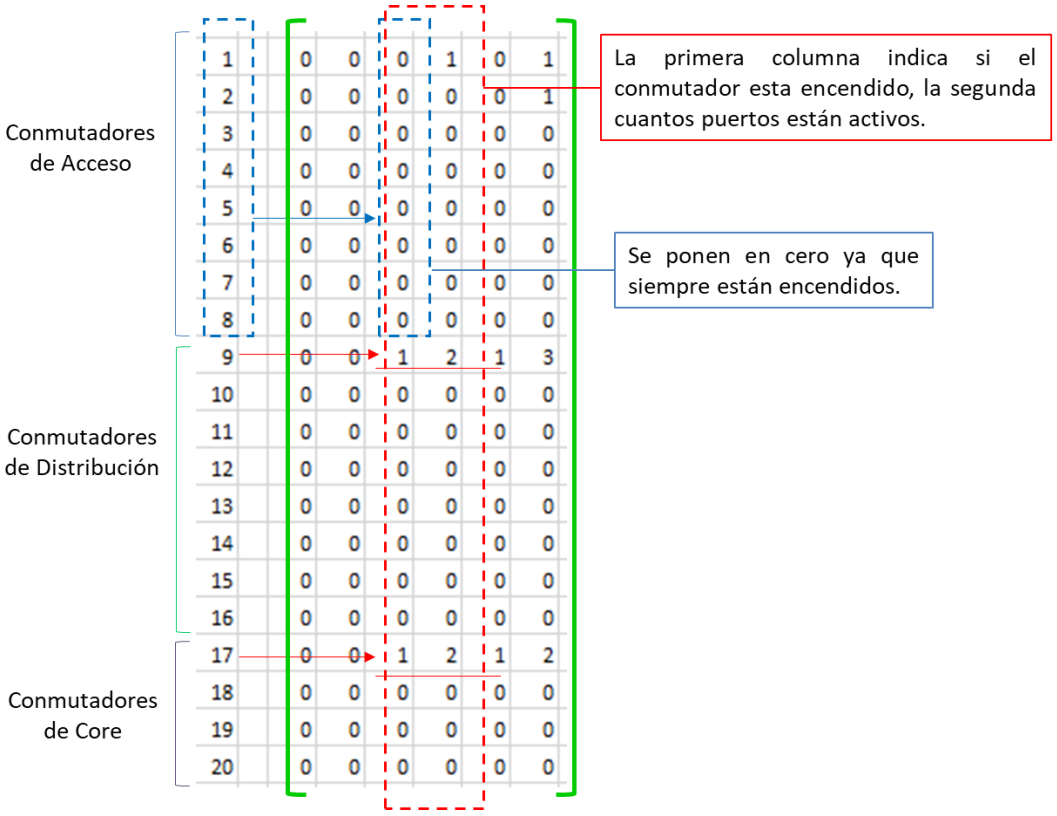


Figura 3.18: Matriz de cambios en la topología.

Esa matriz es exportada a un archivo en formato csv, para utilizarse en el cálculo energético.

5. Matriz de Actualización de Tablas de Flujo (matTablaFlujo): esta matriz se utiliza para almacenar los conmutadores (nodos) y los puertos que deben actualizarse en las tablas de flujo. Esta matriz se crea una vez que el algoritmo estableció la ruta a seguir para el nuevo flujo de datos. La matriz tendrá tantas filas como el doble de conmutadores (nodos) que debe actualizar; y las columnas necesarias para identificar: nodo, puerto de entrada, puerto de salida, prioridad, dirección Mac, origen o destino.

En la figura 3.19 podemos ver un ejemplo de una matriz: matTablaFlujo, que actualizara una ruta nueva que incluye a los conmutadores 1, 9 y 17. La Mac origen y/o destino le pertenece a un host de usuario y a un servidor del conjunto de 16 servidores. Por cada conmutador se crean reglas de ida y vuelta.

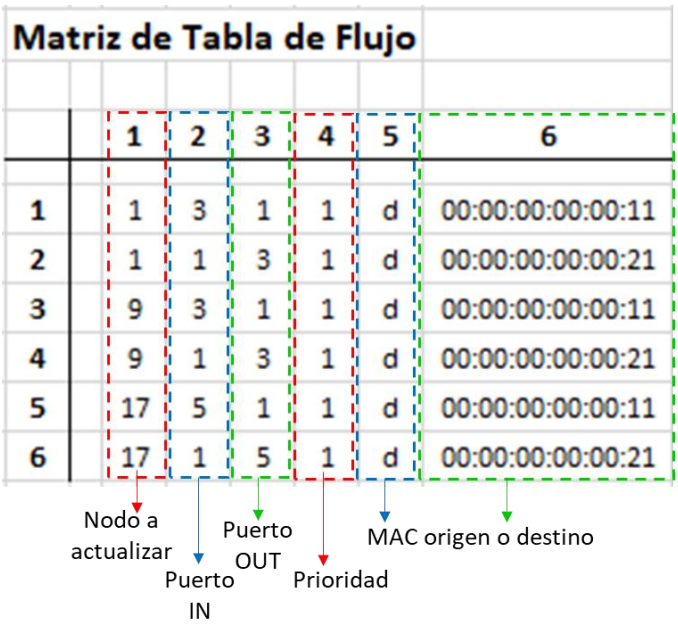


Figura 3.19: Matriz de actualización Tabla de Flujo.

6. Matriz de Trafico Algoritmo A* (matTraficoAStar): esta matriz es un subconjunto de la matriz de tráfico. Se utiliza para el cálculo de la ruta optima utilizando el algoritmo A*. Se tratará en detalle más adelante.

7. Matriz de Nodos de Entrada Algoritmo A* (matNodosAStarIn): esta matriz contiene los nodos y los enlaces que se utilizarán para el cálculo de la ruta óptima, utilizando el algoritmo A*. Además, incluye información sobre los nodos y enlaces como el ancho de banda disponible y otra necesaria para calcular los costos de las rutas. Se tratará en detalle más adelante.

8. Matriz de Nodos de Salida Algoritmo A* (matNodosAStarOut): esta matriz contiene los nodos que forman la ruta óptima hallada utilizando el algoritmo A*. Se tratará en detalle más adelante.

9. Matriz de Direcciones de Servidores (matServidoresDir): esta matriz contiene las direcciones IP y MAC de los servidores, así como los conmutadores y puertos a los que están conectados. Se tratará en detalle más adelante.

3.3.2. Identificación del Origen y Destino

Para identificar el origen y destino de la comunicación utilizaremos una solicitud de establecimiento de enlace. Esta solicitud será enviada por el conmutador monitor hacia el controlador a través de la red de control. El controlador leerá las direcciones IP origen y destino de la solicitud; la dirección origen será validada para verificar si pertenece a un rango valido. Si el origen paso la validación, se validará la dirección destino para verificar que pertenece a la subred de servidores.

En la figura 3.10 podemos observar al conmutador monitor dentro de la topología del centro de datos, este dispositivo como ya se mencionó anteriormente no entra dentro del cálculo energético y está conectado a la red de control. En este proyecto estamos considerando solo un conmutador que actuará como puerta de entrada a la red del centro de datos, dependiendo del tamaño de la red podrían haber más de uno. Sin embargo, para fines prácticos con un conmutador es suficiente para atender el tráfico de este proyecto.

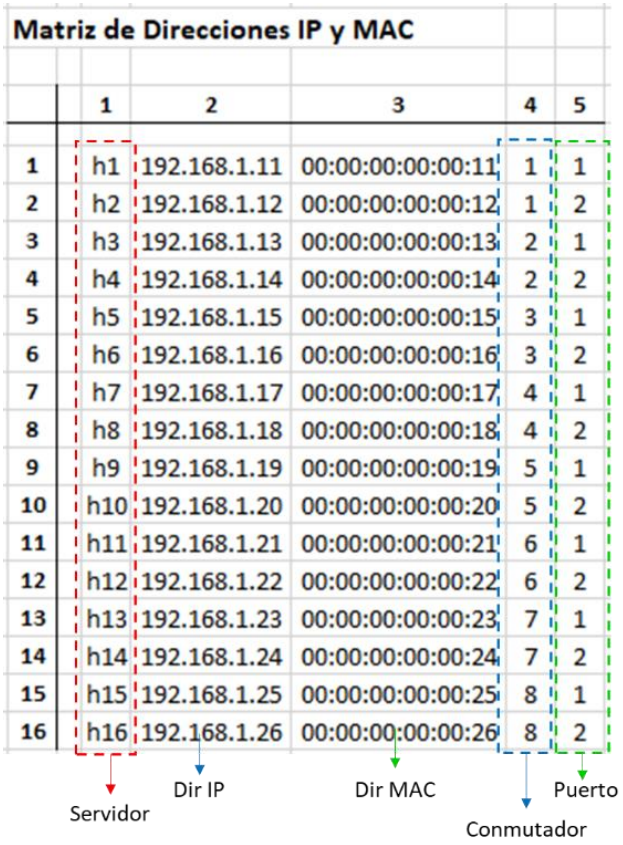
Cuando llega un paquete proveniente de la red de usuarios, el conmutador monitor verifica si hay una ruta en su tabla de flujos; si no encuentra una ruta para ese paquete envía una solicitud de establecimiento de enlace al controlador. Esta solicitud incluye información de capa 02 y capa 03 del origen y el destino. Lo primero que hace el controlador es identificar las direcciones IP origen y destino, y verificar si el origen es válido, paso seguido debe validar si el destino es uno de los servidores. Para esto considera lo siguiente:

- IP Origen: debe pertenecer a la subred de usuarios.
  Rango de IPs de usuarios: 192.168.1.101 – 192.168.1.199

- IP Destino: debe pertenecer a la subred de servidores.
  Rango de IPs de servidores: 192.168.1.11 – 192.168.1.30

Utilizando la dirección IP el controlador identifica el servidor destino que se quiere alcanzar y utilizando la matriz de direcciones de servidores es posible identificar además el conmutador y el puerto al que el servidor destino está conectado. En la

figura 3.20 se puede ver un ejemplo de la matriz: matServidoresDir, para la IP destino 192.168.1.16, el servidor destino es el h6, el conmutador al que está conectado es el número 3 y a través del puerto 2.



	1	2	3	4	5
1	h1	192.168.1.11	00:00:00:00:00:11	1	1
2	h2	192.168.1.12	00:00:00:00:00:12	1	2
3	h3	192.168.1.13	00:00:00:00:00:13	2	1
4	h4	192.168.1.14	00:00:00:00:00:14	2	2
5	h5	192.168.1.15	00:00:00:00:00:15	3	1
6	h6	192.168.1.16	00:00:00:00:00:16	3	2
7	h7	192.168.1.17	00:00:00:00:00:17	4	1
8	h8	192.168.1.18	00:00:00:00:00:18	4	2
9	h9	192.168.1.19	00:00:00:00:00:19	5	1
10	h10	192.168.1.20	00:00:00:00:00:20	5	2
11	h11	192.168.1.21	00:00:00:00:00:21	6	1
12	h12	192.168.1.22	00:00:00:00:00:22	6	2
13	h13	192.168.1.23	00:00:00:00:00:23	7	1
14	h14	192.168.1.24	00:00:00:00:00:24	7	2
15	h15	192.168.1.25	00:00:00:00:00:25	8	1
16	h16	192.168.1.26	00:00:00:00:00:26	8	2

Figura 3.20: Matriz de direcciones de servidores.

A partir de esta información se crean las matrices: matTraficoAStar y matNodosAStarIn, ambas son creadas a partir de las matrices de adyacencias y tráfico que vimos en las estructuras de datos. Estas dos matrices son pasadas al algoritmo que hallará la ruta óptima.

3.3.3. Cálculo de la ruta optima

Una vez identificado el origen y el destino se realizará el cálculo de la ruta óptima para establecer que enlaces serán utilizados por el flujo de datos identificado. Se utiliza un algoritmo de búsqueda heurístico conocido como A*, esta selección se realizó en el marco teórico, ya que para un FatTree con $n < 14$ se mantiene un tiempo de cómputo adecuado y tiene como ventaja el cálculo incremental de la solución lo que mejora su eficiencia.

Como vimos en el marco teórico, el algoritmo A* o A Estrella, calcula la solución óptima en función al costo de la distancia avanzada desde el punto de inicio g(n) y a una función

heurística que mide la distancia del punto actual al punto destino h(n). La función f(n) por lo tanto se calcula así:

$$f_{(n)} = g_{(n)} + h_{(n)} \quad (3.29)$$

Por cada nodo A* evalúa la función f(n) y se quedará con los nodos que presenten el mejor valor de f(n) en el camino al destino. Para calcular esta función es necesario conocer los nodos, los enlaces entre ellos, las rutas posibles y el costo de los enlaces. Esta información es recibida por el algoritmo en las matrices matTraficoAStar y matNodosAStarIn.

En la figura 3.21 vemos un ejemplo de cómo se crea la matriz de tráfico para el algoritmo A*, matTraficoAStar. Una vez identificado el servidor destino, se identifica el conmutador al que este está conectado. Este conmutador será considerado el nodo inicial. Para hacer un ejemplo, el conmutador 01 es el nodo inicial, tomándolo entonces como punto de partida debemos dirigirnos hacia el destino, conmutador 21; en ese sentido identificamos los conmutadores 9, 10 en la capa de distribución y los conmutadores 17, 18, 19 y 20. De esta manera, comprobamos que solo es necesario enviar un subconjunto de la topología FatTree al algoritmo A* y no la matriz completa, lo que nos ahorra tiempo de cálculo y espacio de memoria; estos ahorros son imperceptibles en este proyecto por el número de nodos, pero en redes más grandes si serian importantes.

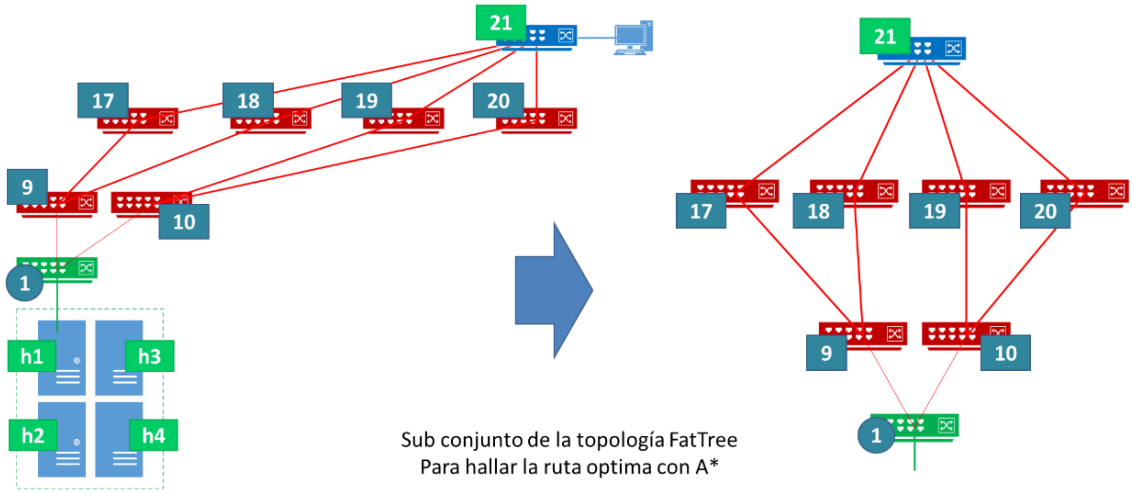


Figura 3.21: Identificación del subconjunto FatTree para el cálculo de la ruta optima.

Sin embargo, enviar solamente los nodos que participarán del cálculo no es suficiente, debemos contar con más información para poder calcular costos de ruta y la función

heurística. Por lo tanto, partimos del subconjunto propuesto en la figura anterior y asignamos un costo a cada enlace. El costo de cada ruta estará en función del tráfico cursado y lo expresaremos como un numero de 0 a 10. Como el algoritmo A* calcula la ruta de menor costo, pero a nosotros nos interesa que los enlaces cursen tráfico hasta que lleguen al límite de su capacidad; entonces 0 significa un enlace congestionado (utiliza toda su capacidad de tráfico) y 10 un enlace sin uso (no está cursando tráfico en ese momento).

En la figura 3.22 hemos agregado el costo a cada ruta posible desde el nodo 1 de inicio, hasta el nodo 21 que es el fin. A cada enlace se le ha asignado un valor de costo en función del tráfico que está cursando en ese momento. Por ejemplo, el enlace 1-10 no cursa tráfico, mientras que el enlace 1-9 está cursando tráfico a la mitad de su capacidad.

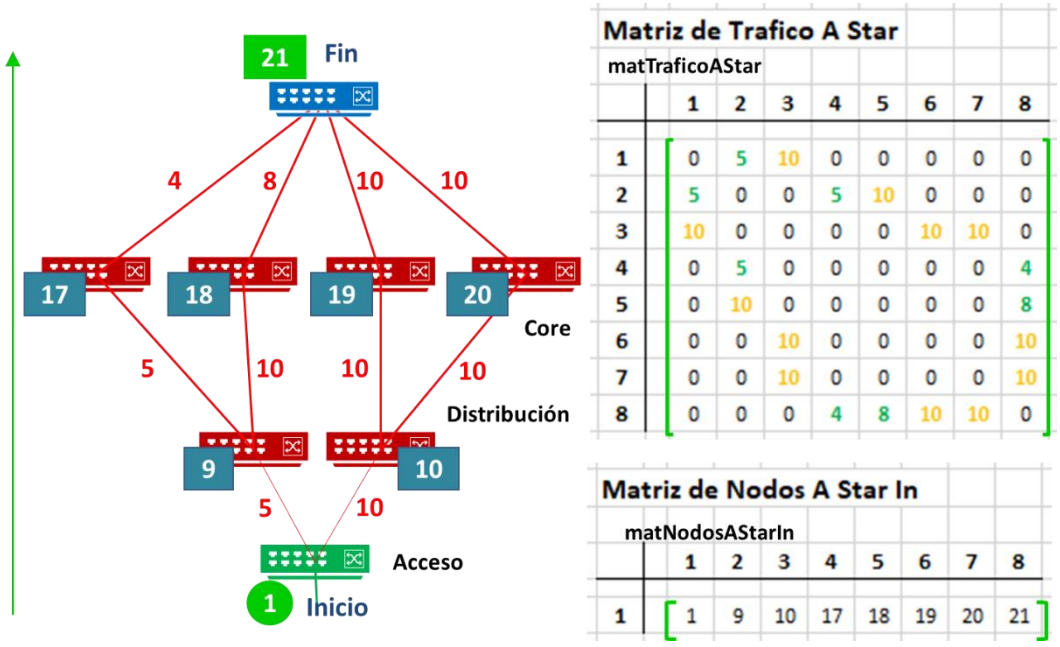


Figura 3.22: Subconjunto FatTree con la asignación de costos. Matrices de entrada para el algoritmo A*

Con esta información se crean las matrices:

matTraficoAStar: subconjunto de la matriz de tráfico, matTrafico. Es una matriz de 8x8, donde cada fila y columna identifica uno de los 8 conmutadores del subconjunto FatTree y su relación de conexión, así como el costo en función del tráfico que están cursando en ese momento. En la figura 3.22 podemos ver la matriz construida en función del subconjunto FatTree del ejemplo de la figura anterior.

matNodosAStarIn: en la matriz matTraficoAStar, para facilitar el cálculo de las rutas los conmutadores son numerado del 1 al 8. La matriz matNodosAStarIn, permite mapear los números reales de los conmutadores con los números asignado en la matriz matTraficoAStar. En la figura 3.22 podemos ver ambas matrices.

Con la información recibida en las matrices explicadas anteriormente el algoritmo A* empieza calculando la función f(n), para cada nodo. Considerando que tomar un subconjunto de FatTree reduce considerablemente el número de nodos a evaluar, en vez de evaluar 21 nodos solo evaluaremos 8, se reducen también considerablemente el número de rutas posibles cada vez que se debe tomar una decisión. El algoritmo A* tomará una decisión de que ruta seguir en 3 niveles:

(1) Escogerá entre dos rutas para llegar al nivel de acceso.
(2) Escogerá entre 4 rutas para llegar al nivel de distribución.
(3) Escogerá entre 4 rutas para llegar al nivel de core.

Adicionalmente en cada elección solo hay un sentido que seguir por lo que no será necesario volver a evaluar nodos una vez que se ha avanzado un nivel. Todo esto facilita el cálculo de la ruta optima.

Una vez seleccionada la ruta el algoritmo A* nos entrega la matriz de salida matNodosAStarOut, en esta matriz están los nodos ordenados en secuencia, siguiendo la ruta encontrada por el algoritmo A*. En la figura 3.23 vemos un ejemplo del resultado, la ruta está formada por los nodos 1, 9, 17 y 21.

Figura 3.23: Nodos en la ruta seleccionada. matNodosAStarOut.

El algoritmo A* se implementa en el script rutasAStar.py.

3.3.4. Actualización de Tablas de Flujo

El resultado de ejecutar el algoritmo nos entrega una matriz de enlaces activos, a partir de esta matriz el controlador puede identificar que conmutadores deberán actualizar sus tablas de flujo y que puertos deben estar activos. Como vemos el ejemplo de la figura 3.24 en color amarillo esta la ruta escogida por el algoritmo y los conmutadores a los que se debe actualizar la tabla de flujo. En verde observamos los enlaces y conmutadores que ya estaban activos y en rojo los alcances y conmutadores que están apagados.

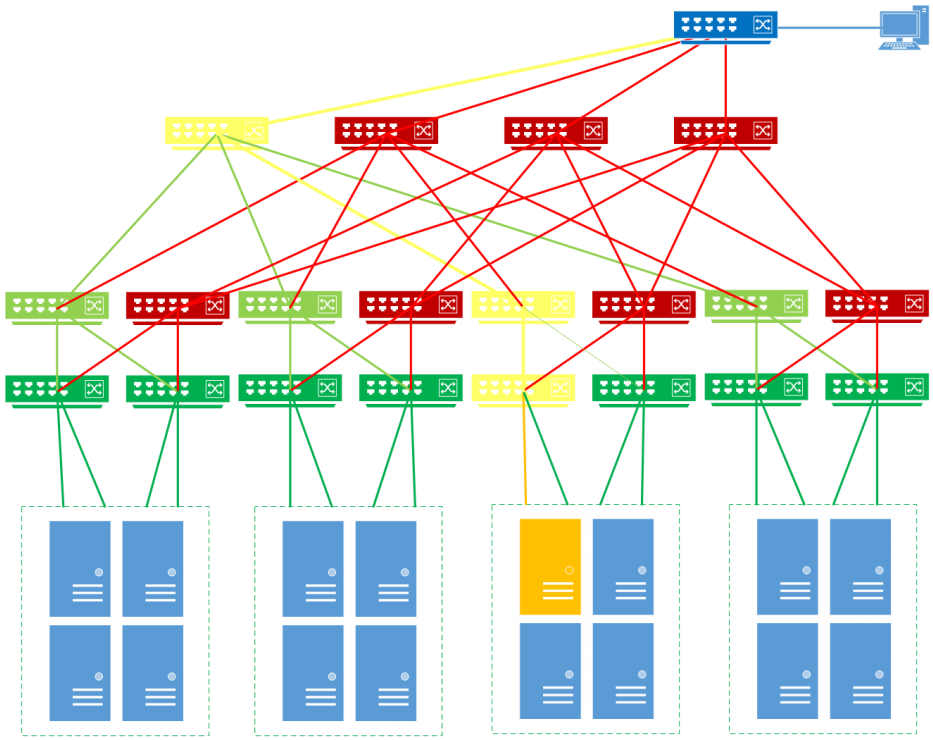


Figura 3.24: Nuevo enlace en amarillo por agregar al ΔTopología final.

El controlador actualiza las tablas de flujo con la siguiente función creada en python:

```
#Esta función nos permite ingresar un nuevo flujo a los conmutadores
def add_flow(self, datapath, priority, match, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    # Construimos el flujo y lo enviamos al conmutador
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,match=match,
instructions=inst)
    datapath.send_msg(mod)
```

El resultado final es un subconjunto de FatTree (ΔTopología), en la figura 3.25 podemos observar en color verde el subconjunto de FatTree que se mantendrá activo por un momento específico y en rojo está el subconjutno de FatTree que debe permanecer desactivado. Utilizando este subconjunto se calcula la potencia consumida en ese momento utilizando el modelo de consumo energético.

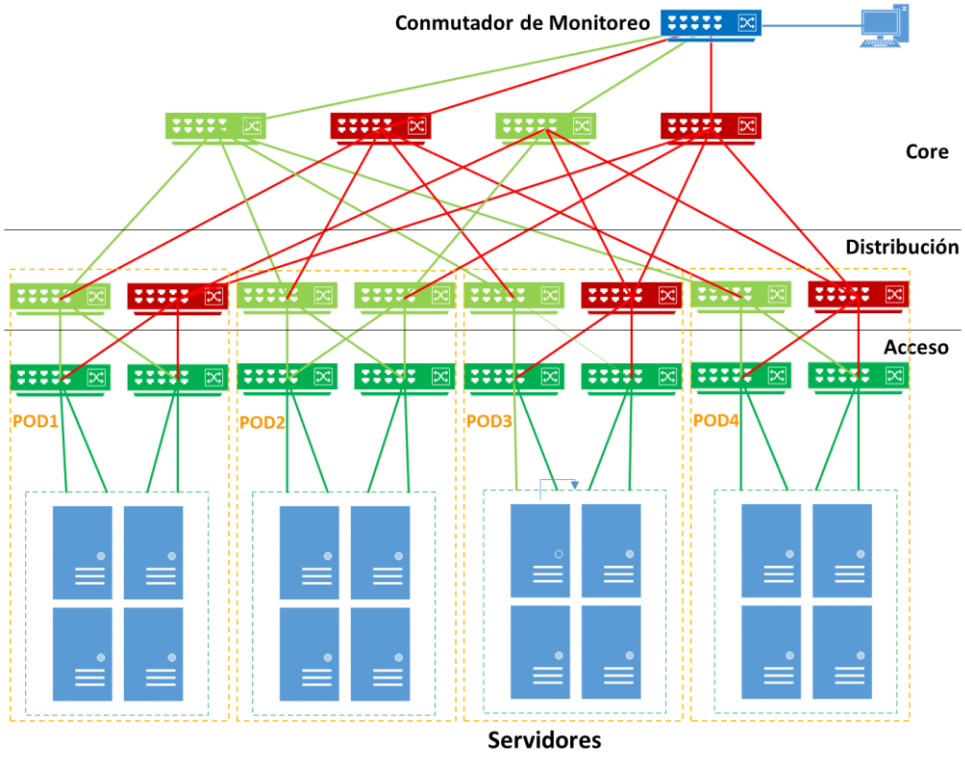


Figura 3.25: ΔTopología – subsoncjunto de FatTree activo en verde.

3.4. Bloque de Simulación

Una vez que está definida la topología, el modelo energético y la lógica de control, que incluye el cálculo de la ruta optima, así como, el encendido y apagado de equipos; procederemos con la simulación de la red para demostrar el funcionamiento del sistema de control energético.

Se simula la topología FatTree utilizando como plataforma mininet, un emulador de redes que soporta el protocolo OpenFlow y la conexión con un controlador. Se ingresarán flujos de tráfico y se variará la topología lógica de la red de acuerdo con los enlaces en uso que permitan canalizar estos flujos. Se recolectarán los datos

3.4.1. Plataforma de Simulación

Como ya mencionamos la herramienta que utilizaremos será principalmente MiniNet, a partir de esta herramienta podemos simular una red SDN con las características descritas en los pasos previos. Y el controlador RYU, que como se mencionó en el marco teórico, lo utilizaremos por su capacidad para soportar pyhton, OpenFlow y por su flexibilidad cuando se trabaja de manera experimental. Sin embargo, necesitaremos otras herramientas que nos permitirán realizar la simulación y ejecutar los scripts en el controlador. Aquí mencionamos el kit de herramientas con las que vamos a trabajar:

- Mininet 2.3.0 -> simulador de redes SDN
- Ryu 4.34 -> controlador SDN basado en pyhton
- Python 3.9 -> Lenguaje de programación, se utiliza para crear los scripts en el controlador y para automatizar la creación de la red en mininet.
- Ubuntu Server 20.04.1 -> Sistema Operativo donde se ejecutan mininet y Ryu.
- Oracle VirtualBox 6.1 -> Máquina virtual donde se ejecuta Ubuntu Server como host de mininet y el controlador SDN Ryu.
- Putty y WinScp -> para establecer conexiones seguras SSH con la MV que ejecuta mininet y con el controlador.
- Visual Studio Code 1.55.0 -> IDE para el desarrollo de los scripts en pyhton.
- OCTAVE 6.2.0 -> Es un leguaje científico de programación, orientado a realizar cálculos matemáticos y al cálculo con matrices. Es una opción libre al MATLAB.

Las simulaciones se desarrollarán sobre una computadora con las siguientes características:

- Lenovo V14-ARE Series.
- Procesador AMD Ryzen 5 2.4 GHz.
- Sistema Operativo Windows 10 de 64 bits.
- Memoria RAM de 16 GB.

Oficialmente en la página de mininet no hay características mínimas de hardware para su instalación, pero si las dos opciones de instalación propuestas:

- Descargar una MV con Ubuntu y mininet ya instalado.
- Descargar las fuentes de mininet y compilarlas e instalarlas en la distro de Linux de tu preferencia.

Se ha seleccionado la primera opción por ser la más rápida para tener una versión de mininet instalada y operativa. Se seleccionó la versión mininet 2.30 y Ububtu 2.04 por ser, ambas, versiones actualizadas y con soporte.

Mininet se ejecuta en una máquina virtual de Ubuntu con Virtual box, como vemos en la figura 3.26. El servidor se ejecuta sin modo gráfico, por lo que todo se gestiona en modo de comandos usando el CLI de Linux.

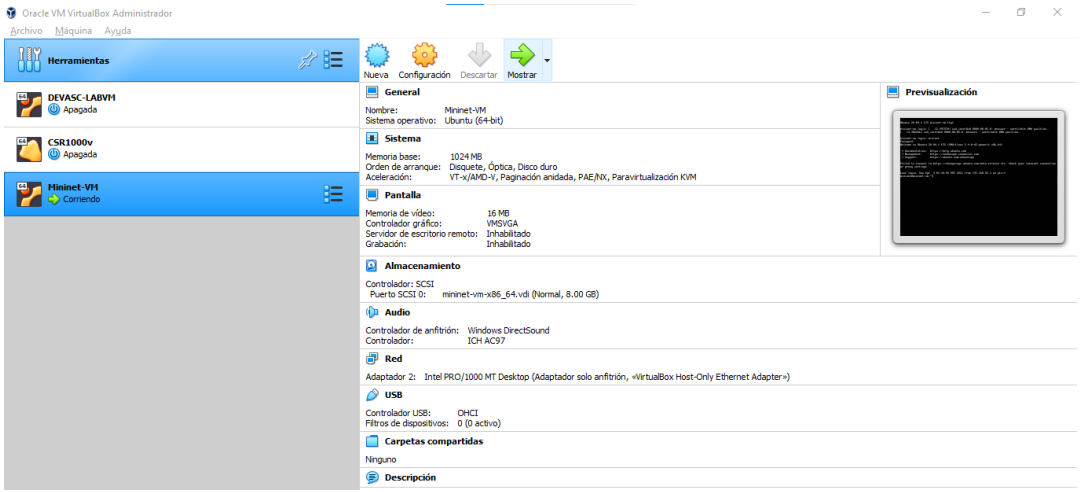


Figura 3.26: Mininet -Ubuntu - VirtualBox

Para acceder al servidor y poder abrir diferentes ventanas de comandos utilizaremos Putty, esta herramienta permite una conexión segura a través del protocolo SSH. Un ejemplo en la figura 3.27

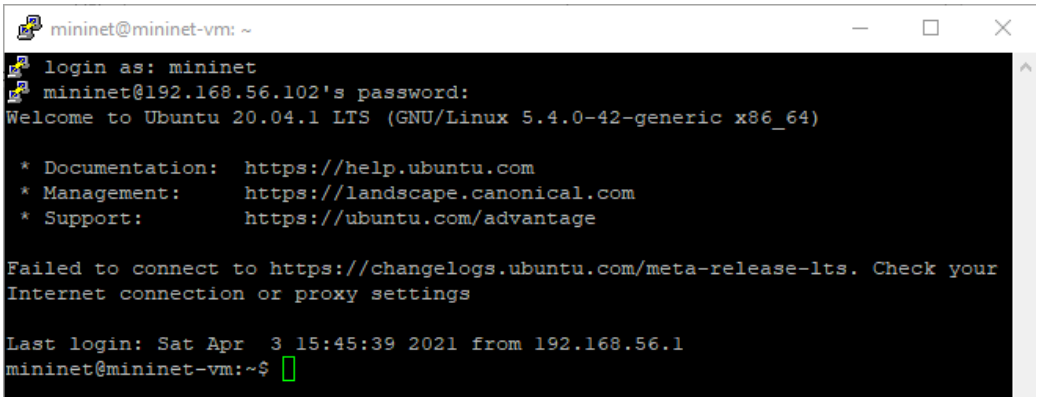


Figura 3.27: Accediendo con Putty al servidor Ubuntu

Finalmente necesitamos intercambiar archivos con la máquina virtual de Ubuntu, estos archivos son los scripts en Python que utilizaremos para construir la red y gestionar los

dispositivos. Para intercambiar archivos utilizaremos WinSCP, utiliza conexiones seguras a través del protocolo SSH para intercambiar archivos, tiene una interfase muy fácil de arrastrar y soltar fácil de utilizar, como se ve en la figura 3.28.

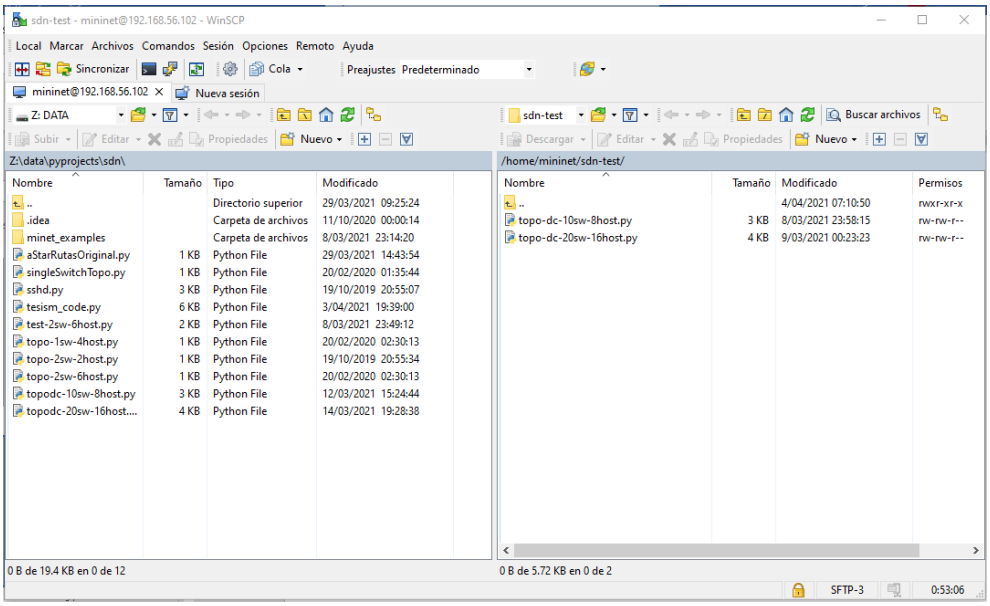


Figura 3.28: Accediendo al sistema de archivos de Ubuntu con WinSCP

Para el desarrollo de los scripts en Python utilizamos un IDE, que es un entorno de desarrollo integrado que nos facilita la creación, depuración y mantenimiento del código en Python. En este caso en la figura 3.29 podemos observar el entorno de trabajo de Visual Studio Code utilizado en este proyecto.

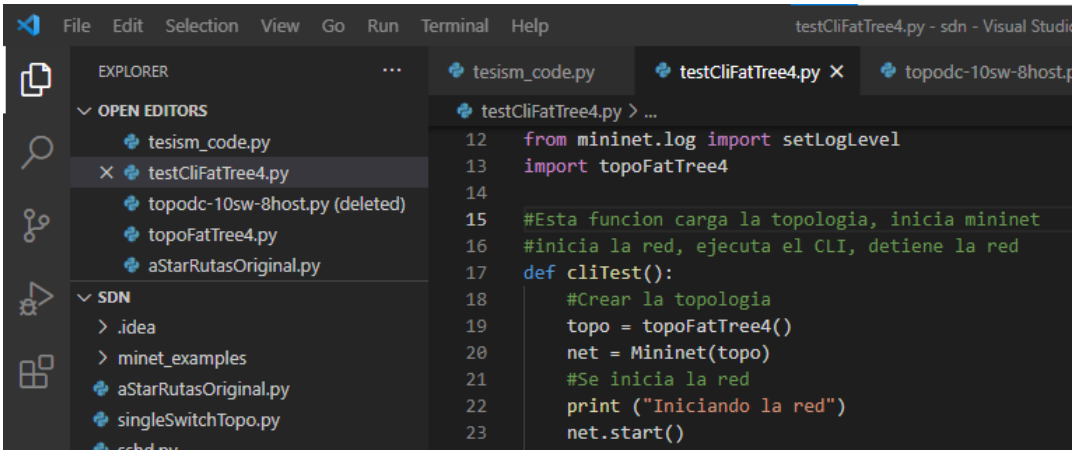


Figura 3.29: Desarrollo de scripts en pyhton con Visual Studio Code

Una vez establecida la plataforma pasamos a realizar la simulación.

3.4.2. Simulación de la Red

Una vez que todas las herramientas están listas para la simulación abrimos 3 interfaces de línea de comandos o CLIs. Esto se hace iniciando la máquina virtual con el servidor Ubuntu que contiene a mininet y abriendo dos sesiones con Putty conectadas al servidor Ubuntu. Como vemos en la figura 3.30 la pantalla 01 es una sesión en putty para ejecutar mininet, la pantalla 02 es otra sesión con putty donde ejecutaremos el controlador RYU y la pantalla 03 es el CLI de la MV con Ubuntu donde ejecutaremos los comandos Linux necesarios.

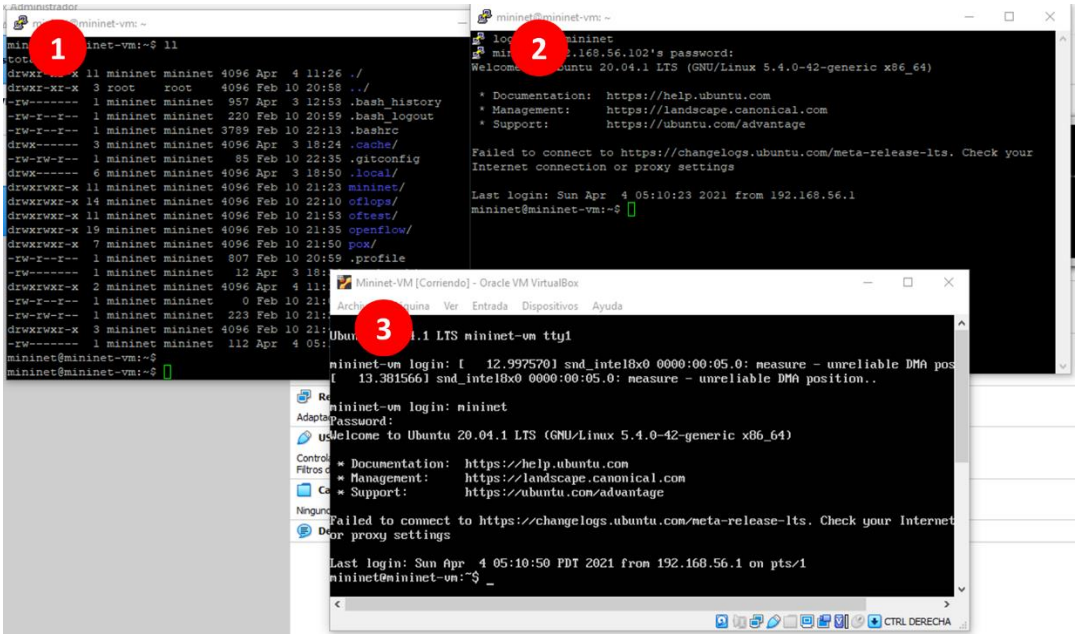


Figura 3.30: Los 3 CLI para controlar la simulación.

En el directorio sdn-tesis se almacenan los scripts en pyhton, estos son creados desde la maquina real con Visual Studio Code y luego enviados a la MV con Ubuntu utilizando WinSCP.

Los pasos para realizar la simulación son los siguientes:

1) Cargamos la topología FatTree n=4 y ejecutamos mininet.
2) Ejecutamos el controlador RYU con el script de monitoreo y gestión de rutas.
3) Inyectamos tráfico hacia los servidores.
4) Guardamos los cambios de topología para el posterior cálculo energético.

Para crear en mininet la topología FatTree n=4 con la que se está trabajando se utiliza un script en python: topoFatTree4.py. En la figura 3.31 tenemos un extracto del script

topoFatTree4.py, creamos una clase con el mismo nombre, con la función build se crea la topología: primero se crean los conmutadores nombrados desde s1 a s21, luego los hosts nombrados desde h1 hasta h17, al momento de crearse cada host se le asigna una dirección IP. Finalmente se crean los enlaces, para cada enlace se define que numero de puerto se utilizará en cada dispositivo a ambos extremos del enlace.

Figura 3.31: Script: topoFatTree4.py.

El comando en mininet para cargar la topología es:

$sudo mn --custom=./sdn-tesis/topoFatTree4.py --topo=topotFatTree4

Sin embargo, no hay un controlador que les diga que hacer a los conmutadores por lo que es necesario decirle a mininet que debe conectarse con el controlador Ryu. En uno de los terminales abiertos con Putty ejecutamos el controlador RYU. Para ver si la topología FatTree n=4 esta correcta ejecutaremos el controlador con un script que implementa un conmutador simple y básico.

Ejecutamos el controlador con el comando:

$ryu-manager --verbose ryu.app.simple_switch

El resultado lo vemos en la figura 3.2, el comando --verbose nos muestra los eventos que se activan:

```
mininet@mininet-vm:~$ ryu-manager --verbose ryu.app.simple_switch
loading app ryu.app.simple_switch
loading app ryu.controller.ofp_handler
instantiating app ryu.app.simple_switch of SimpleSwitch
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK SimpleSwitch
  CONSUMES EventOFPPacketIn
  CONSUMES EventOFPPortStatus
BRICK ofp_event
  PROVIDES EventOFPPacketIn TO {'SimpleSwitch': {'main'}}
  PROVIDES EventOFPPortStatus TO {'SimpleSwitch': {'main'}}
  CONSUMES EventOFPEchoReply
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPErrorMsg
  CONSUMES EventOFPHello
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPPortStatus
  CONSUMES EventOFPSwitchFeatures
```

Figura 3.32: Ejecución del controlador ryu en modo --verbose.

Utilizando otro terminal con Putty ejecutamos mininet, sin embargo, agregaremos el comando --controller remote, con este comando mininet buscará conectarse con un controlador remoto en la dirección de loopback 127.0.0.1 y buscará los puertos 6653 y 6633. En este caso Ryu escucha en el puerto 6653. Ejecutamos el comando:

$sudo mn –controller remote --custom=./sdn-tesis/topoFatTree4.py --topo=topotFatTree4

```
mininet@mininet-vm:~$ sudo mn --controller remote --custom ./sdn-tesis/topoFatTree4.p
y --topo=topoFatTree4
*** Creating network
*** Adding controller
Connecting to remote controller at 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19 s20 s21
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (h4, s2) (h5, s3) (h6, s3) (h7, s4) (h8, s4) (h9, s5) (h10
, s5) (h11, s6) (h12, s6) (h13, s7) (h14, s7) (h15, s8) (h16, s8) (h17, s21) (s1, s9)
 (s1, s10) (s2, s9) (s2, s10) (s3, s11) (s3, s12) (s4, s11) (s4, s12) (s5, s13) (s5,
s14) (s6, s13) (s6, s14) (s7, s15) (s7, s16) (s8, s15) (s8, s16) (s9, s17) (s9, s18)
(s10, s19) (s10, s20) (s11, s17) (s11, s18) (s12, s19) (s12, s20) (s13, s17) (s13, s1
8) (s14, s19) (s14, s20) (s15, s17) (s15, s18) (s16, s19) (s16, s20) (s17, s21) (s18,
 s21) (s19, s21) (s20, s21)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17
*** Starting controller
c0
*** Starting 21 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19 s20 s21 ...
*** Starting CLI:
mininet>
```

Figura 3.33: Ejecución de mininet con la topología FatTree

En la figura 3.33 vemos el resultado de la ejecución del comando anterior, la topología FatTree N=4 es desplegada, primero se crean los hosts de h1 al h16, que son los servidores y el h17 que representa a los usuarios. Luego se crean los conmutadores s1 hasta s21 y después se crean los enlaces. También se configuran los hosts y los conmutadores. Se inicia el controlador C0, este controlador hace referencia al controlador ryu que acabamos de iniciar.

Ya tenemos la topología FatTree cargada en mininet y el controlador Ryu ejecutando el script simple_switch que permite un comportamiento básico de los conmutadores. Estamos ejecutando Ryu con este script básico, como dijimos anteriormente, para probar que nuestra topología esté funcionando correctamente, antes de hacer las pruebas de control de rutas y energía. Para verificar que la topología está bien desplegada, ejecutamos los siguientes comandos en mininet:

Mininet> nodes

En la figura 3.34, se puede ver que tenemos los 16 hosts (servidores de h1 a h16) y los 21 conmutadores activados (s1 a s21).

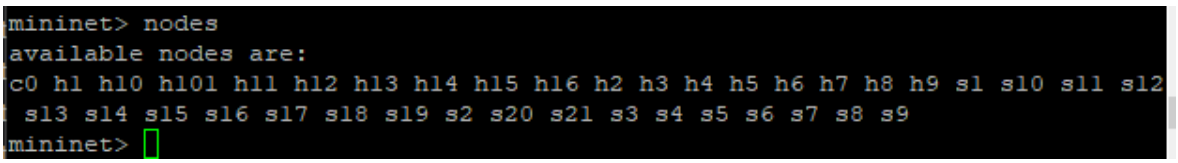


Figura 3.34: Salida del comando nodes.

Mininet> links

En la figura 3.35 podemos ver los enlaces creados, la imagen no muestra la salida completa ya que es muy extensa. Se muestra cada enlace creado entre dos dispositivos indicando la interfase y el estado. En este caso todos los enlaces están OK.

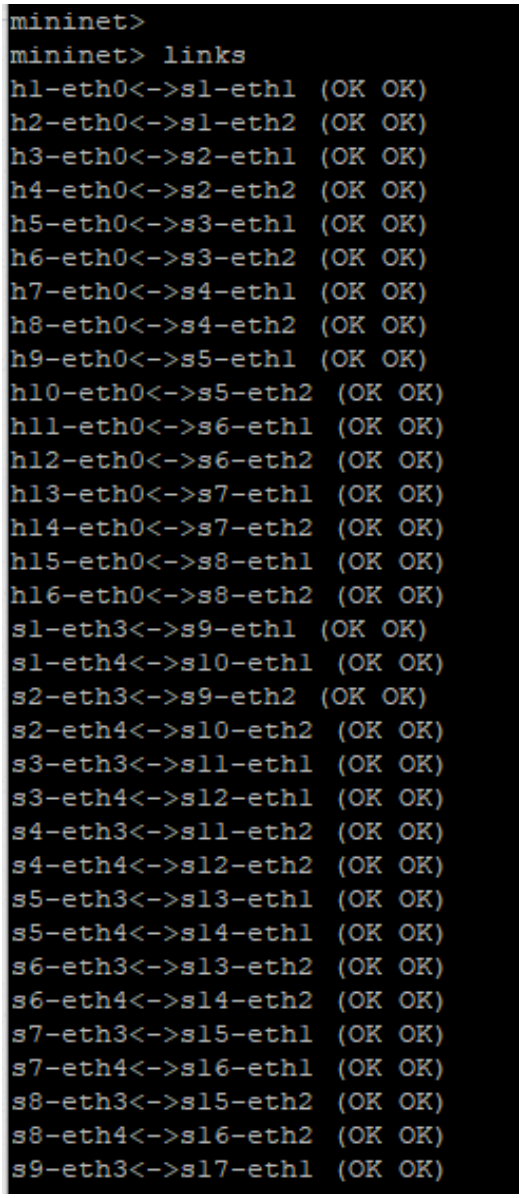


Figura 3.35: Salida del comando links.

Mininet> ports

En la figura 3.36 se puede ver el resultado de ejecutar el comando *ports*. Se listan los puertos activos de cada conmutador desde el s1 al s21 y a través de que interfase funciona. Por ejemplo, para el conmutador s1: la interface de loopback (lo) funciona en el puerto 0, el puerto 1 en eth1, el puerto 2 en eth2, el puerto 3 en eth3 y el puerto 4 en eth4.

```
mininet>
mininet> ports
s1 lo:0 s1-eth1:1 s1-eth2:2 s1-eth3:3 s1-eth4:4
s2 lo:0 s2-eth1:1 s2-eth2:2 s2-eth3:3 s2-eth4:4
s3 lo:0 s3-eth1:1 s3-eth2:2 s3-eth3:3 s3-eth4:4
s4 lo:0 s4-eth1:1 s4-eth2:2 s4-eth3:3 s4-eth4:4
s5 lo:0 s5-eth1:1 s5-eth2:2 s5-eth3:3 s5-eth4:4
s6 lo:0 s6-eth1:1 s6-eth2:2 s6-eth3:3 s6-eth4:4
s7 lo:0 s7-eth1:1 s7-eth2:2 s7-eth3:3 s7-eth4:4
s8 lo:0 s8-eth1:1 s8-eth2:2 s8-eth3:3 s8-eth4:4
s9 lo:0 s9-eth1:1 s9-eth2:2 s9-eth3:3 s9-eth4:4
s10 lo:0 s10-eth1:1 s10-eth2:2 s10-eth3:3 s10-eth4:4
s11 lo:0 s11-eth1:1 s11-eth2:2 s11-eth3:3 s11-eth4:4
s12 lo:0 s12-eth1:1 s12-eth2:2 s12-eth3:3 s12-eth4:4
s13 lo:0 s13-eth1:1 s13-eth2:2 s13-eth3:3 s13-eth4:4
s14 lo:0 s14-eth1:1 s14-eth2:2 s14-eth3:3 s14-eth4:4
s15 lo:0 s15-eth1:1 s15-eth2:2 s15-eth3:3 s15-eth4:4
s16 lo:0 s16-eth1:1 s16-eth2:2 s16-eth3:3 s16-eth4:4
s17 lo:0 s17-eth1:1 s17-eth2:2 s17-eth3:3 s17-eth4:4 s17-eth5:5
s18 lo:0 s18-eth1:1 s18-eth2:2 s18-eth3:3 s18-eth4:4 s18-eth5:5
s19 lo:0 s19-eth1:1 s19-eth2:2 s19-eth3:3 s19-eth4:4 s19-eth5:5
s20 lo:0 s20-eth1:1 s20-eth2:2 s20-eth3:3 s20-eth4:4 s20-eth5:5
s21 lo:0 s21-eth1:1 s21-eth2:2 s21-eth3:3 s21-eth4:4
mininet>
```

Figura 3.36: Salida del comando ports.

Ahora se pasa a comprobar que el controlador está funcionando adecuadamente. Para esto primero lo apagamos temporalmente. Una vez apagado hacemos la siguiente prueba:

Mininet> pingall

El comando pingall hace un ping entre todos los hosts disponibles en la topología. La figura 3.37 muestra el resultado, las X indican que no hay respuesta desde ningún host.

```
mininet>
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X X X X X X X X X X X X X X
h2 -> X X X X X X X X X X X X X X X X
h3 -> X X X X X X X X X X X X X X X X
h4 -> X X h3 X X X X X X X X X X X X X
h5 -> X X X X X X X X X X X X X X X X
h6 -> X X X X X X X X X X X X X X X X
h7 -> X X X X X X X X X X X X X X X X
```

Figura 3.37: Salida del comando pingall.

Este resultado se debe a que al no estar activo el controlador las tablas de flujo de los conmutadores están vacías, esto lo comprobamos con el comando:

Mininet> dpctl dump-flows

En la figura 3.38 se observa la salida del comando dpctl dump-flows, como se observa las tablas de flujo de los 21 conmutadores están vacías.

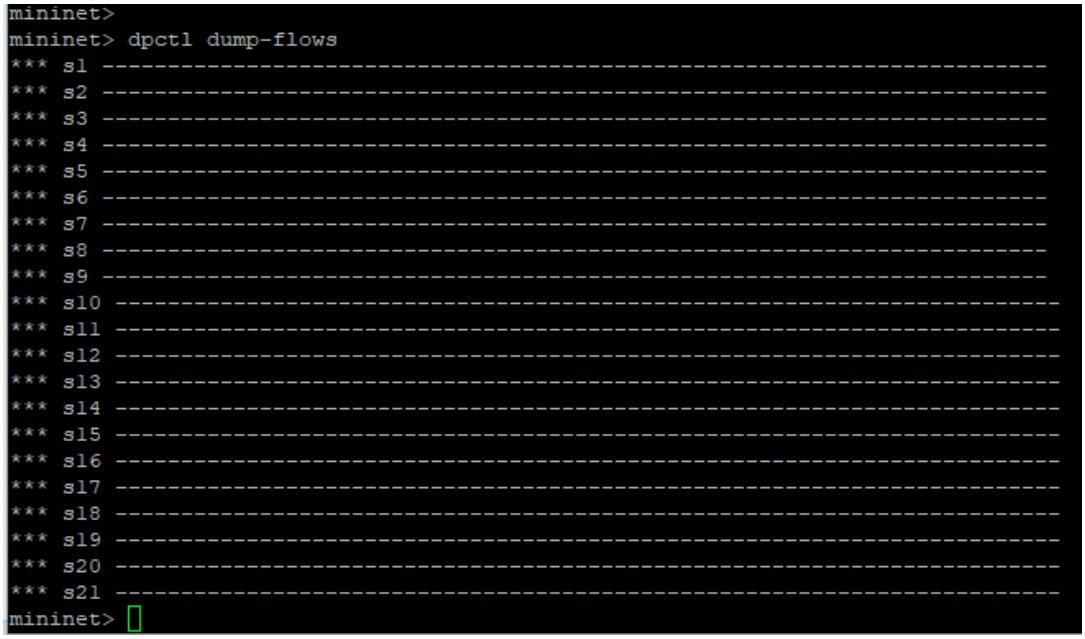


Figura 3.38: Salida del comando: dpctl dump-flows.

Se procede a reiniciar el controlador, para que la salida no sea muy extensa y no demore mucho hacemos la prueba haciendo un ping entre el servidor1 (h1) y el servidor2 (h2).

Mininet> h1 ping h2

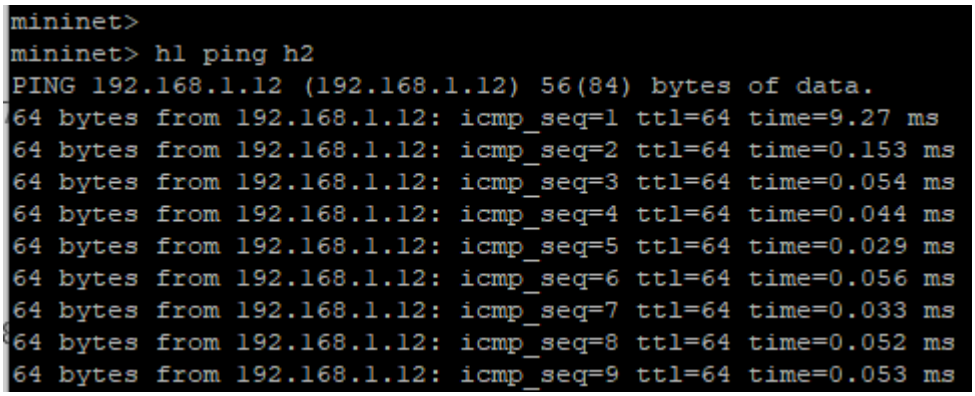


Figura 3.39: Salida del comando: ping entre h1 y h2.

Al ejecutar el comando ping, los conmutadores reciben los paquetes y no tienen instrucciones que seguir en sus tablas de flujo. Si embargo, como el controlador esta activo

les da las indicaciones de qué hacer con los paquetes. A través del protocolo OpenFlow envían las indicaciones a sus tablas de flujo. Lo que les permite reenviar los paquetes, en la figura 3.39 podemos ver el resultado de ejecutar le comando ping, esta vez podemos ver que llego al destino y este emitió una respuesta.

Para ver las tablas de flujo actualizadas utilizamos el siguiente comando:

Mininet> dpctl dump-flows

En la figura 3.40 vemos como el conmutador 1 (s1) ha actualizado sus tablas de flujo, la tabla de flujo tiene dos entradas:

- La primera entrada tiene la regla: si ingresa (in_port) tráfico por el puerto 1 (s1-eth1) con mac origen 00:00:00:00:00:11 y destino mac 00:00:00:00:00:12, acción debe ser sacarlo por el puerto 2 (actions=output:"s1-eth2").
- La segunda entrada tiene la regla: si ingresa (in_port) tráfico por el puerto 2 (s1-eth2) con mac origen 00:00:00:00:00:12 y destino mac 00:00:00:00:00:11, acción debe ser sacarlo por el puerto 1 (actions=output:"s1-eth1").

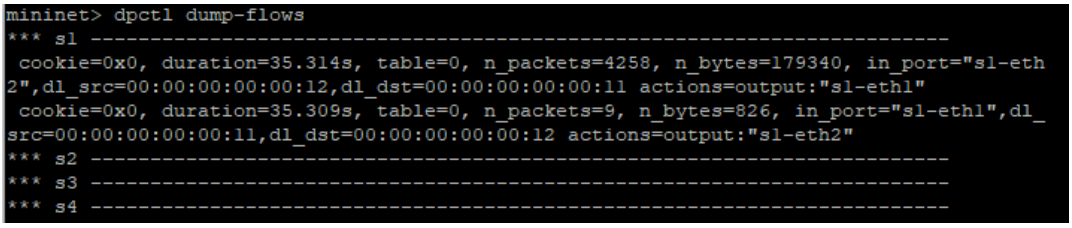


Figura 3.40: Tablas de flujo en el conmutador 01.

Cuando hacemos ping, en el terminal donde estamos ejecutando el controlador es posible ver alguna información que nos proporciona el controlador. En este caso como se ejecutó con la opción --verbose podemos ver información adicional. En la figura 3.41 vemos la salida que se produce cuando ejecutamos un ping entre dos dispositivos, algo que podemos resaltar es que cada vez que llega un paquete al controlador se genera el evento: EventOFPacketIn, se debe tomar nota de esto, ya que es este evento el que permite tomar acciones cada vez que el controlador recibe un paquete y se tiene la necesidad de buscar una ruta para ese flujo.

```
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 1 00:00:00:00:00:12 ff:ff:ff:ff:ff:ff 2
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 2 00:00:00:00:00:12 ff:ff:ff:ff:ff:ff 5
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 2 00:00:00:00:00:16 00:00:00:00:00:12 3
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 1 00:00:00:00:00:16 00:00:00:00:00:12 5
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 1 00:00:00:00:00:12 00:00:00:00:00:16 2
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 2 00:00:00:00:00:12 00:00:00:00:00:16 5
```

Figura 3.41: Salida de eventos en el controlador Ryu.

Ahora que se ha probado que la topología y el controlador están funcionando adecuadamente. Se ejecutará en el controlador el script *controlRutasOptimas.py*, que realizará el cálculo de la ruta optima en función del tráfico recibido. Este script utilizará el evento *EventOFPacketIn* para recibir mensajes de los conmutadores que reciban un paquete y si no tienen un match en su tabla de flujo, el controlador lo procese.

```
55      #Este evento se lanza cada vez que un paquete le llega al conmutador proveniente
56      #de un conmutador
57      @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
58
59      #Con esta funcion manejamos el paquete que ingresa al controlador
60      def _packet_in_handler(self, ev):
61          msg = ev.msg
62          datapath = msg.datapath
63          ofproto = datapath.ofproto
64          parser = datapath.ofproto_parser
65
66          # Obtenemos el Datapath ID para identificar los conmutadores.
67          dpid = datapath.id
68          self.mac_to_port.setdefault(dpid, {})
69
70          # Analizamos el paquete recibido: obtenemso la direccion origen y destino.
71          pkt = packet.Packet(msg.data)
72          eth_pkt = pkt.get_protocol(ethernet.ethernet)
73          dst = eth_pkt.dst
74          src = eth_pkt.src
```

Figura 3.42: Script: controlRutasOptimas.py.

En la figura 3.42 se puede observar el evento openflow: ofp_event.EventOFPacketIn, una vez que el evento se lanza se ejecuta la función _packet_in_handler, que es la función que nos permite manjar un paquete entrante a través de ev.msg. En la figura 3.43 se observa el código que ejecuta los diferentes pasos para el cálculo de la ruta optima y en la figura 3.44 se observa el código para la actualización de las tablas de flujo.

```
77          #Aqui empieza el codigo de la Tesis
78          #1. Se verifica el destino del paquete.
79          i = 0
80          while i < 16:
81
82              if matHostDir(i,0) == dst:
83                  nodoInicio = matHostDir(i,0)
84                  break
85              i += 1
86
87          #2. Se calcula el subconjunto de la matriz de trafico
88          matTraficoAStarIn = getTraficoAStar(nodoInicio)
89
90          #3. Se aplica el Algoritmo
91          matNodosAStarOut = rutasAStar(matTraficoAStar)
92
93          #4. Se carga la tabla de flujos por cada nodo en la ruta
94          for nodo in matNodosAStarOut:
95
96              actions = [parser.OFPActionOutput(out_port)]
97
98              actualizacion=getUltimoCambio(nodo)
```

Figura 3.43: Extracto de controlRutasOptimas.py.

```
93          #4. Se carga la tabla de flujos por cada nodo en la ruta
94          for nodo in matNodosAStarOut:
95
96              actions = [parser.OFPActionOutput(out_port)]
97
98              actualizacion=getUltimoCambio(nodo)
99
100             #5. se instala el flujo.
101             if out_port != ofproto.OFPP_FLOOD:
102             match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
103             self.add_flow(datapath, 1, match, actions)
104
105             #6. se construye el mensaje packet_out y se envía.
106             out = parser.OFPPacketOut(datapath=datapath,
107                                       buffer_id=ofproto.OFP_NO_BUFFER,
108                                       in_port=in_port, actions=actions,
109                                       data=msg.data)
110             datapath.send_msg(out)
```

Figura 3.44: Actualización de tablas de flujo en controlRutasOptimas.py.

Iniciamos el controlador con el comando:

$ryu-manager ryu.app.controlRutasOptimas

Ya no utilizamos el comando --verbose, para evitar un conjunto de salidas innecesarias.

Una vez iniciado el controlador se ejecuta el script: testElasticTree4.py, este script nos permite automatizar todos los pasos: lanzar mininet, cargar la topología, iniciar la red, cargar tráfico a los servidores, realizar los cálculos de rutas y detener la red. El comando es:

$ sudo -E python ./sdn_test/testElasticTree4.py

En la figura 3.45 podemos ver como se automatizan las pruebas con este script, primero se carga un array matTestIP con los hosts (servidores) que serán accedidos durante la prueba. Este array tiene 60 nombres de host, ya que representarán la activación de host por minuto, para poder completar la hora que debe durar la prueba.

testElasticTree4.py > testElasticTree

```
17  def testElasticTree():
18      #Creamos la matriz de prueba:
19      matTestIP["h1","h1","h1","h3","h8","h8","h10","h10","h10","h10",'
20
21      #Crear la topologia FatTree n=4
22      topo = topoFatTree4()
23      controller=remote
24      net = Mininet(topo,controller)
25
26      #Se inicia la red
27      print ("Iniciando la red")
28      net.start()
29      time.sleep(2)
30
31      #Creamos un ping hacia cada direccion de la matriz de prueba
32      print ("Iniciando Pruebas de ping")
33      hostOrig = net.get('h27')
34      hostOrig_IP = hostOrig.IP()
35      for ipd in matTestIP:
36          hostDest = net.get(idp)
37          hostDest_IP = hostDest.IP()
38          testCmd = 'ping -c10 %s' % hostDest_IP
39          hostOrig.cmd(testCmd)
40          print("Ping a: ", hostDest_IP)
41          time.sleep(10)
42
43      #Se detiene la red
44      print ("Deteniendo la red")
45      net.stop()
46
```

Figura 3.45: Script: testElasticTree4.py.

Luego se carga la topología topoFatTree4(), que ya hemos visto y probado anteriormente, seguido definimos que usaremos el controlador remoto, el cual ya se está ejecutando en la dirección 127.0.0.1 y puerto 6653. Se inicia la red y aplicamos un retardo

de 2 segundos para asegurar que la topología cargue. Se inician las pruebas de tráfico iniciando un ping a cada host del array matTestIP. Después de recorrer el array de pruebas se detiene la red.

Cada vez que el controlador valida un cambio de topología se crea una entrada en el archivo *matCambiosTopo.csv*, esta entrada consta de dos columnas: la primera indica que conmutadores están encendidos, la segunda cuantos puertos de ese conmutador están encendidos. La figura 3.46 muestra el código que realiza los guardados en el archivo por cada cambio en topología.

```
112     #7. Guardamos la actualizacion en el archivo matCambiosTopo.csv
113     f = open("matCambiosTopo.cvs", "a")
114     f.write(actualizacion)
115     f.close()
```

Figura 3.46: Script: testElasticTree4.py.

Este archivo matCambiosTopo.csv es el punto de partida para el siguiente capítulo, a partir de los cambios de topología calcularemos el ahorro de energía según el modelo energético planteado.

CAPITULO IV
ANALISIS Y RESULTADOS

En el presente capítulo, se hace uso de la herramienta Octave para analizar los datos de la simulación y calcular el ahorro de energía.

4.1. Resultados de la Investigación

Para realizar el análisis de resultados hemos utilizado la herramienta OCTAVE versión 6.2.0, se han creado scripts ".m" que procesarán los datos obtenidos en el capítulo anterior. En la figura 4.1 vemos un resumen de los componentes de la topología FatTree, y como han sido numerados; en color verde los componentes siempre activos y en rojo los de estado variable. Estos son los elementos que serán evaluados para establecer el porcentaje de ahorro energético posible, por lo tanto, para los cálculos solo se considerarán los 20 conmutadores y los 84 puertos.

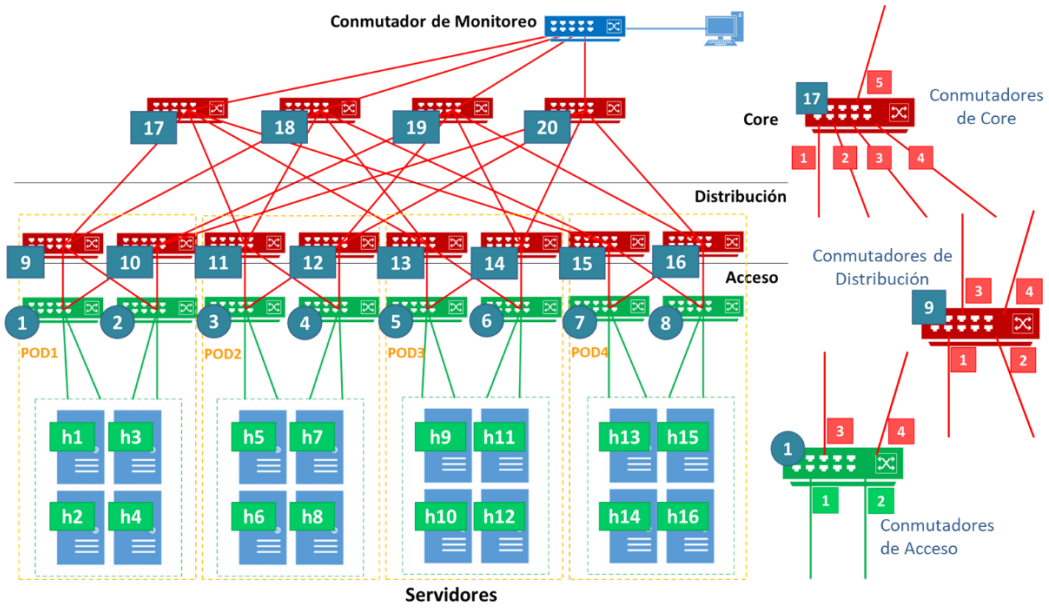


Figura 4.1: Topología FatTree. Componentes.

Podemos observar que del 1 al 8 están los conmutadores de acceso, del 9 al 16 los conmutadores de distribución y del 17 al 20 los conmutadores de Core. Los puertos se numeran en cada conmutador del 1 al 4. Los puertos 1 - 2 apuntan hacia el sur de la

topología y los puertos 3 - 4 hacia el norte. En el caso de los conmutadores de core 1 -4 apuntan hacia el sur y el puerto 5 hacia el norte, que representa la red de usuarios.

Tal como se mencionó en el capítulo 03 los arreglos de matrices utilizados para el almacenamiento de información de gestión de topología se mantienen en memoria durante la simulación. Cada vez que hay un cambio de topología, este se almacena en la matriz de cambios matCambiosTopo, esta matriz tiene 20 filas y n columnas; donde cada fila representa un conmutador y cada par de columnas representa: 1 si el conmutador este encendido y 2 cuantos puertos tiene encendidos.

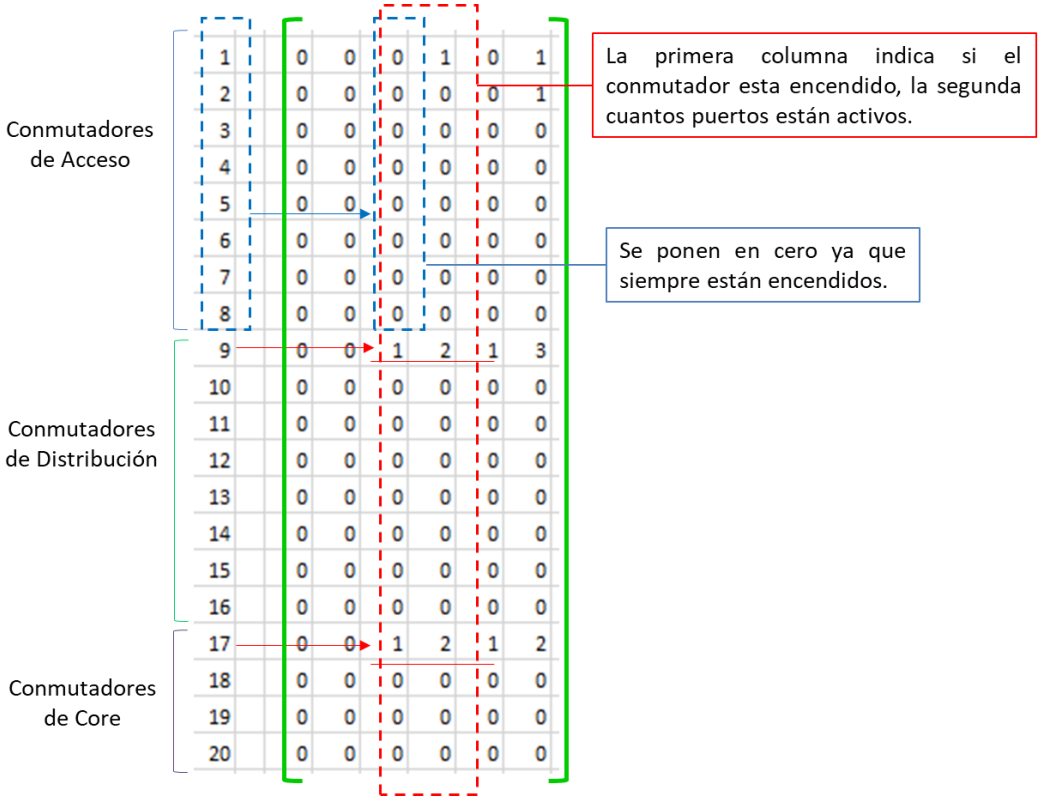


Figura 4.2: descripción de la matriz matCambiosTopo.

Como observamos en la figura 4.2 en la matriz de cambios de topología las 20 filas representan los 20 conmutadores, los 8 primeros son los conmutadores de acceso, los 8 siguientes los conmutadores de distribución y los últimos 4 son los conmutadores de core.

Para realizar el análisis la matriz de cambios de topología se guarda como un archivo separado por comas; de esta manera se carga al script en OCTAVE que utiliza la fórmula de cálculo de potencia que encontramos utilizando el modelo energético.

Para el cálculo de potencia primero consideraremos un escenario donde está funcionando la topología FatTree completa, todos los conmutadores están encendidos y todos los enlaces están activos. Luego realizaremos los cálculos en el escenario ElasticTree considerando primero solo los dispositivos que siempre permanecen encendidos, los 8 conmutadores en la capa de acceso con dos puertos cada uno. Finalmente se iniciarán flujos de datos hacia los servidores y se irán activando los componentes de red bajo demanda. Se calculará la potencia en cada caso y se comparará con la potencia calculada para la topología FatTree completa.

Utilizaremos los valores de potencia para los conmutadores y puertos considerados en el capítulo 03; también consideramos el número de componentes que formarán una topología FatTree, n=4. Los cargamos en el script de OCTAVE como vemos en la figura 4.3.

```
13  #Valores energéticos predefinidos en Watts
14  Pch = 151;
15  Pi = 3.83;
16  Pa = 4;
17
18  #Valores de la topologia Fat-Tree
19  n=4;
20  numSWTotal = (5/4)*n^2;
21  numSWacceso = (n^2)/2;
22  numSWdistro = (n^2)/2;
23  numSWCore = (n^2)/4;
24  numServidores = (n^3)/4;
25  numPortTotal = (5/4)*n^3 + numSWCore;
26
27  #Escenario Fat-Tree
28  PotFatTree = numSWTotal*Pch + numPortTotal*Pa;
29
30  #Calculamos la potencia consumida por los dispositivos siempre encendidos
31  #Para hallar el consumo fijo de potencia
32  #Conmutadores acceso:
33  numSWfijo = numSWacceso;
34  numPuertosFijo = numServidores;
```

Figura 4.3: OCTAVE carga de parámetros iniciales.

Para la topología FatTree completa consideraremos los 20 conmutadores y los 84 puertos. Con estos datos calculamos la potencia en watts que consume FatTree al funcionar, independiente de cuanto tráfico se esté cursando a través de sus enlaces o cuantos servidores estén siendo accedidos en ese momento.

Luego calcularemos el consumo fijo de potencia para la topología flexible de ElasticTree. Según la figura 4.2, los dispositivos que funcionan permanentemente son los

computadores de acceso 1 al 8 y los puertos 1 – 2 de los mismos. Por lo tanto, se consideran para el cálculo de potencia fija: 8 conmutadores y 16 puertos. Es decir, los dispositivos que se mantendrán encendidos así no estén cursando tráfico alguno.

Para el tercer escenario se considera una situación ideal de ElasticTree, en la que los 16 servidores están recibiendo tráfico moderado. El algoritmo de cálculo de rutas activaría los conmutadores y los enlaces en color verde que observamos en la figura 4.4.

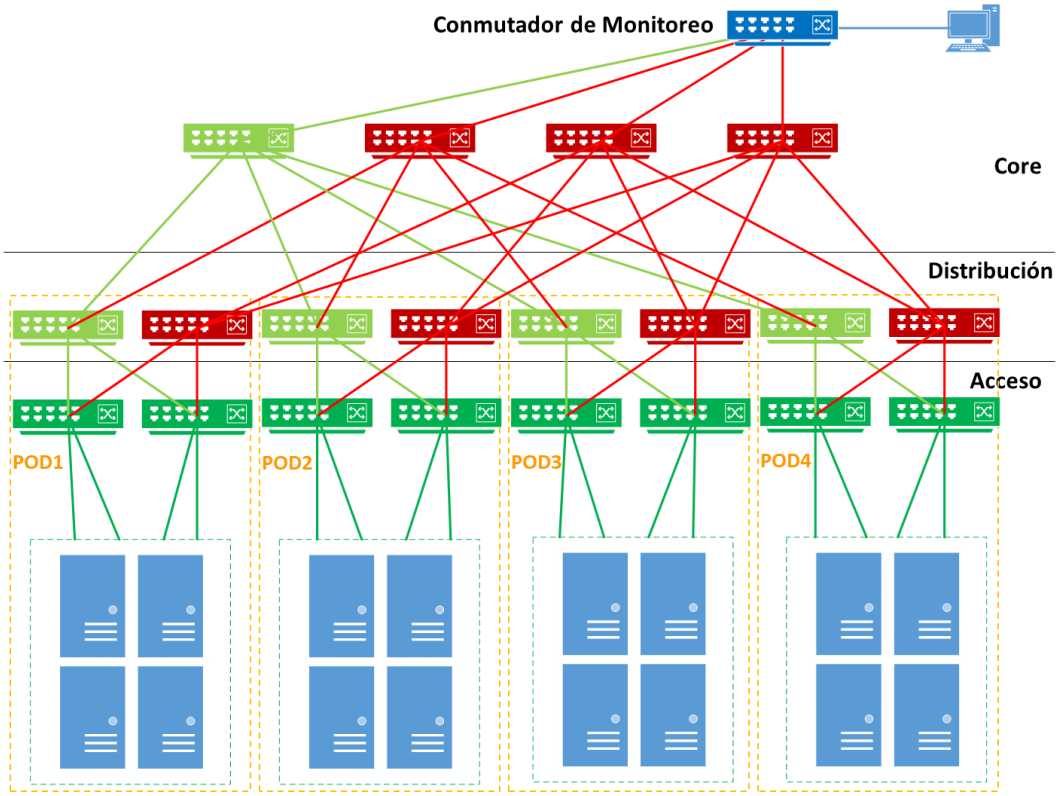


Figura 4.4: Escenario ElasticTree ideal.

Para calcular la potencia consumida en este escenario debemos agregar al cálculo de Potencia fija, la potencia consumida por los dispositivos encendidos: 5 conmutadores y 25 enlaces adicionales A esta última potencia la consideramos potencia variable ya que dependerá del conjunto conmutadores y enlaces encendidos. Entonces resumiendo las fórmulas del modelo energético y considerando que n = 4 tendríamos:

Para FatTree:

$$P_{FatTree} = \frac{5}{4}n^2 \times P_{chasis} + \left(\frac{5}{4}n^3 + \frac{n^2}{4}\right) \times P_r = 3356.00\ Watts \quad (4.\ 1)$$

Para la potencia fija ElasticTree:

$$P_{Fija} = \frac{5}{4}n^2 \times P_{chasis} + \left(\frac{5}{4}n^3 + \frac{n^2}{4}\right) \times P_r = 1272.00\ Watts \quad (4.\ 2)$$

Para la Potencia ElasticTree escenario ideal.

$$P_{ElasticTree} = n_{Sw} \times P_{chasis} + n_{puertos} \times P_r = 2127.00\ Watts \quad (4.\ 3)$$

Ahorro de energía:

$$AhorroEnergia = (1 - \frac{P_{ElasticTree}}{P_{FatTree}}) \times 100 = 36.62\ \% \quad (4.\ 4)$$

Cargamos estos 3 escenarios en OCTAVE y obtenemos el resultado que observamos en la figura 4.5 que corrobora los cálculos anteriores.

```
Ventana de comandos
Inicamos el calculo energético...

Escenarios Básicos

Escenario 01: Fat-Tree
-Conmutadores encendidos: 20
-Puertos encendidos: 84
-Potencia Total:   3356.00 Watts

Escenario 02: Elastic-Tree sin trafico
-Conmutadores encendidos: 8
-Puertos encendidos: 16
-Potencia Fija:   1272.00 Watts
-Potencia Variable:      0.00 Watts
-Potencia Total:   1272.00 Watts

Escenario 03: Elastic-Tree con trafico ideal
-Conmutadores encendidos: 13
-Puertos encendidos: 41
-Potencia Fija:   1272.00 Watts
-Potencia Variable:    855.00 Watts
-Potencia Total:   2127.00 Watts

Ahorro de energía Elastic-Tree con trafico ideal vs Fat-Tree
-Ahorro de energía:     36.62 %
```

Figura 4.5: Cálculo de potencia. Escenarios básicos.

Ahora utilizaremos un scritp en OCTAVE para graficar la curva de consumo de potencia a medida que se van conectando recursos de red. El escenario será utilizando ElasticTree, partimos de la situación de tráfico cero, cuando ningún servidor está siendo utilizado. En ese punto solo consumen energía los dispositivos que están siempre activos,

el consumo fijo de potencia es de 1 272.00 Watts, tal como se calculó anteriormente. Gradualmente se irá accediendo a los servidores iniciando en el h1 hasta el h16; y en consecuencia se irán habilitando los recursos necesarios, conmutadores, enlaces y puertos, hasta llegar al escenario ElasticTree ideal de la figura 4.4.

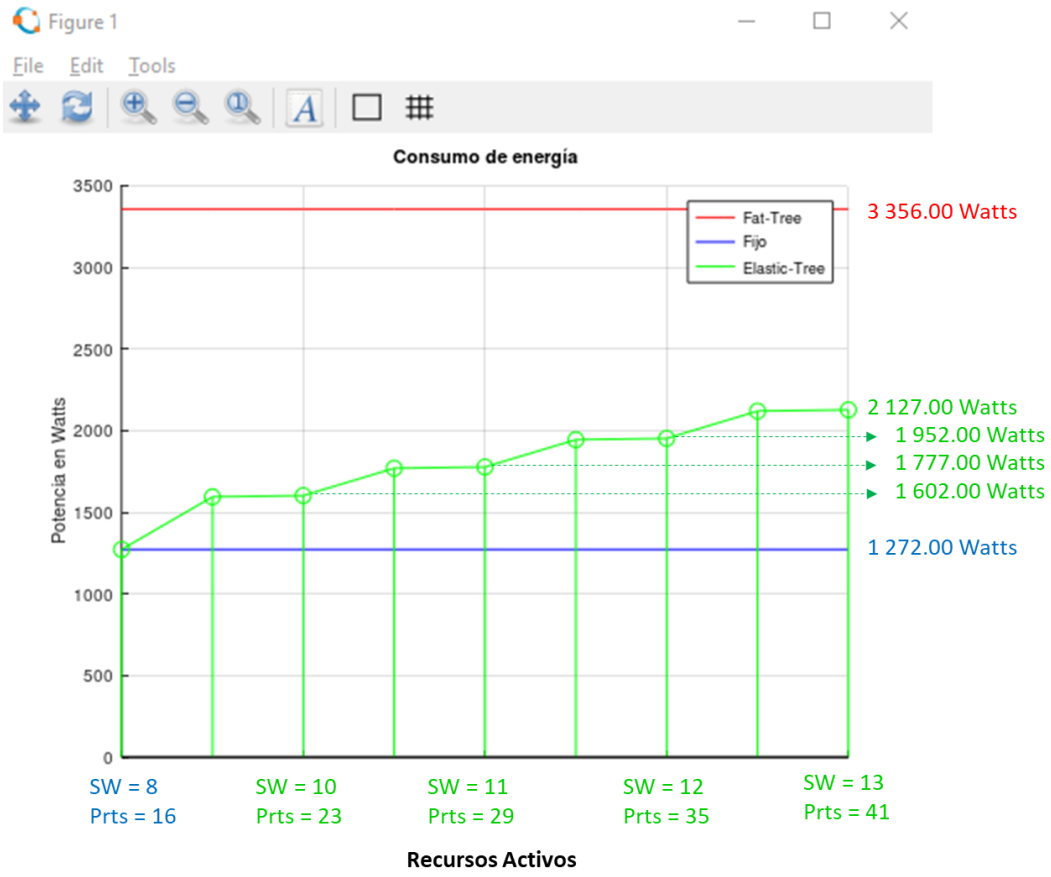


Figura 4.6: Variación de potencia vs número de recursos encendidos.

En la figura 4.6 podemos observar cómo se va incrementando el consumo de potencia a medida que se van encendiendo los equipos de la red del centro de datos. Iniciamos con el consumo fijo de 1 272.00 watts antes de que se inicie el intercambio de tráfico de datos, al llegar a 1 602.00 Watts tenemos 10 conmutadores encendidos y 23 puertos. A los 1 777.00 watts alcanzamos los 11 conmutadores y los 29 puertos. Y finalmente llegamos a los 2 127.00 Watts con 13 conmutadores y 41 puertos encendidos.

En este punto ideal de funcionamiento observamos que con FatTree el consumo seria 3 356.00 Watts, mientras que con ElasticTree es de 2 127.00 Watts.

Después de hacer los cálculos con el escenario básico que implica el consumo de la topología FatTree completo y el de ElasticTree ideal, podemos concluir que efectivamente se presenta un ahorro en el consumo de energía, 33.6% aproximadamente. Sin embargo, una prueba en un escenario más realista es necesaria. Adicionalmente para medir el consumo energético se debe cursar tráfico hacia los servidores durante, por lo menos, una hora de tal manera que podremos expresar el consumo en Watts-hora (Wh).

La simulación en mininet se ejecutó durante una hora cursando tráfico de manera gradual hacia los servidores, la matriz matCambiosTopo, se descarga en un archivo separado por comas. En la Figura 4.7 vemos un pequeño extracto del archivo que contiene los datos almacenados en la matriz de cambios topológicos. Cada 2 columnas representan el estado de la topología en un momento específico considerando los 20 conmutadores y sus puertos. La matriz se actualiza cada minuto durante una hora por lo que cuenta con 120 columnas.

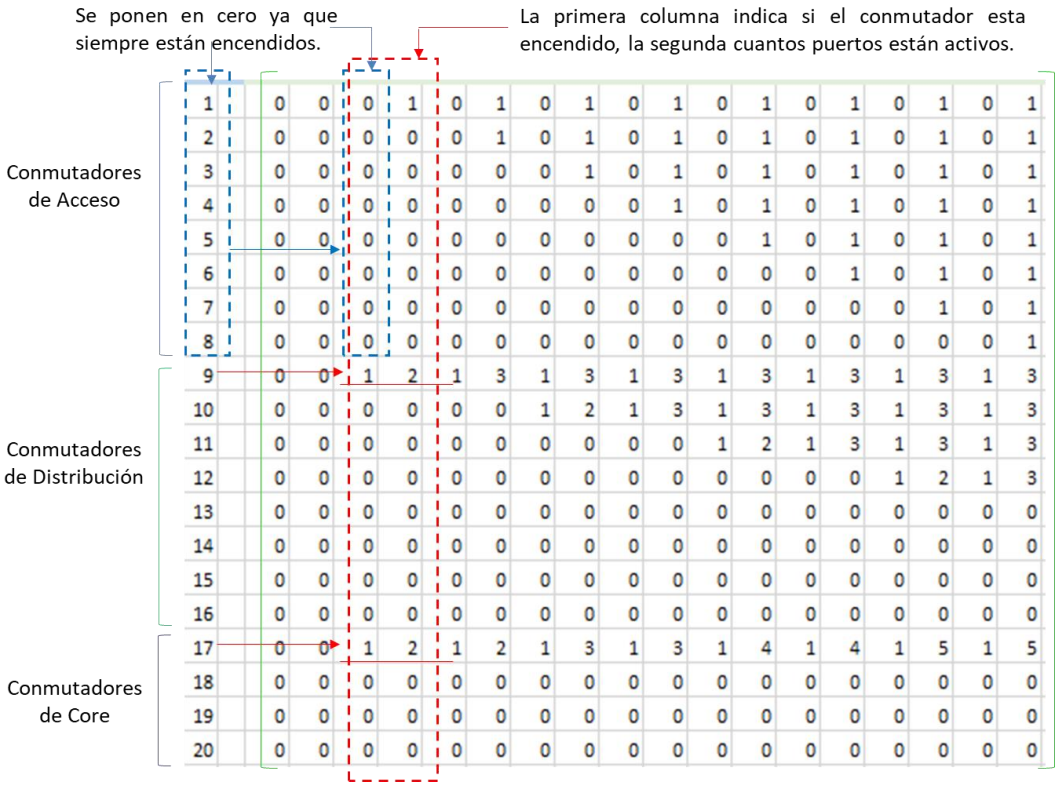


Figura 4.7: Ejemplo de la matriz matCambiosTopo.

Este archivo es cargado primero en excel para y luego ser importado en OCTAVE con los comandos que vemos en la Figura 4.8, de esta manera cargamos el contenido del archivo de Excel en una matriz que llamamos matActivosFlujo en OCTAVE:

```
40  #Archivo que contiene los datos
41  nomArchivo = 'matCambiosTopo.xlsx';
42  nomHojaIni = 'activos_01';
43  nomHojaFlujo = 'activos_03';
44
45  #Cargamos la matriz con el dato de puertos y conmutadores activos
46  pkg load io;
47  matActivosIni = xlsread(nomArchivo,nomHojaIni);
48  matActivosFlujo = xlsread(nomArchivo,nomHojaFlujo);
```

Figura 4.8: Carga del archivo matCambiosTopo.xlsx a OCTAVE.

El script recorre la matriz matActivosFlujo y genera la curva de la Figura 4.9. El eje X está en minutos y el eje Y es la potencia en Watts. La recta azul representa el valor fijo de potencia de los equipos siempre encendidos, en la figura vemos el número de conmutadores y puertos. La línea roja representa la potencia que consume la topología FatTree completa, es decir todos los conmutadores y puertos de los conmutadores encendidos.

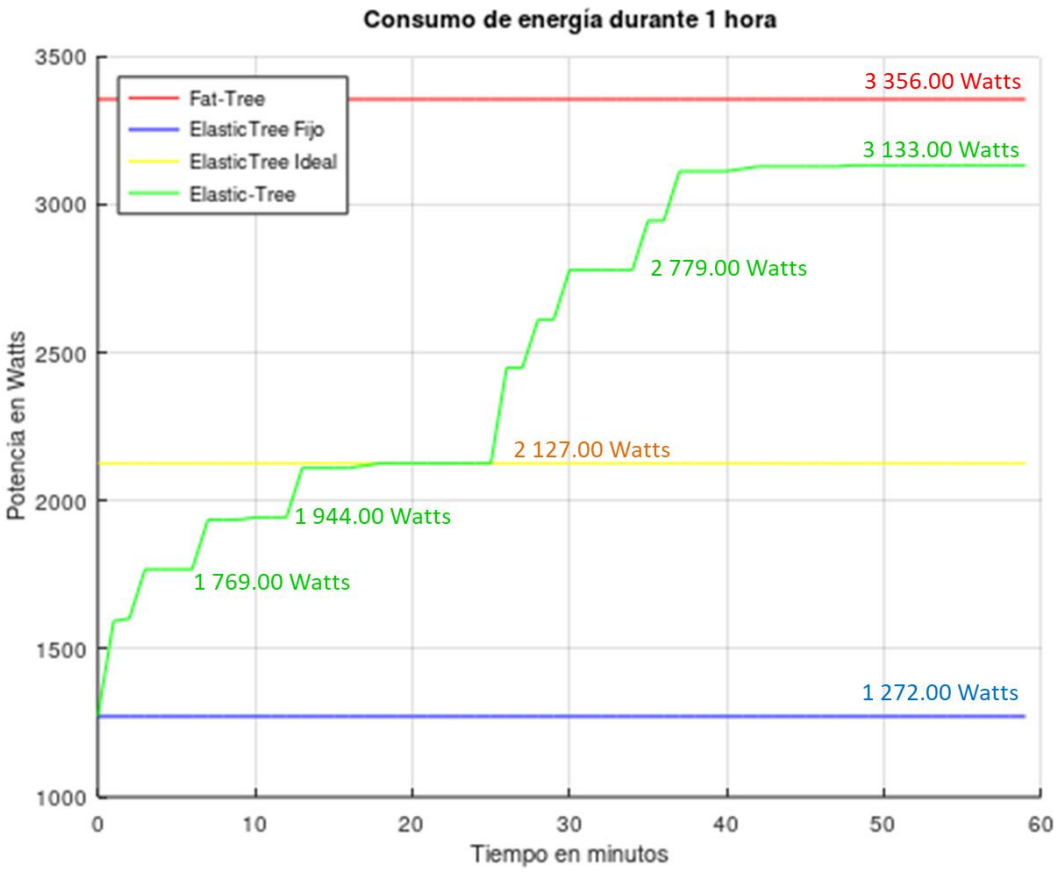


Figura 4.9: Curva de consumo de potencia en una hora.

La línea amarilla es nos da como referencia el consumo de ElasticTree ideal. La curva de color verde representa la potencia variable, que va aumentando a medida se encienden los conmutadores y sus puertos debido a la carga de tráfico que va dirigida hacia los servidores. Podemos observar algunos valores: inicia en 1 272.00 Watts y va subiendo, pasando por 1 769.00 Watts, 2127.00 Watts, 2 779.00 Watts hasta llegar a 3 133 Watts en el minuto 60.

Para el mismo escenario podemos observar otra curva en la Figura 4.10, esta vez observamos el incremento del consumo de potencia a medida que se van activando los conmutadores y puertos. En color amarillo vemos la línea que marca el consumo de la topología ElasticTree ideal, sin embargo, podeos observar que esta simulación sobrepaso ese consumo. Esto se debe a que después de alcanzar a la topología ideal, se siguió con el incrementando del tráfico y estos enlaces se congestionaron; obligándole al sistema a levantar otros caminos como respaldo.

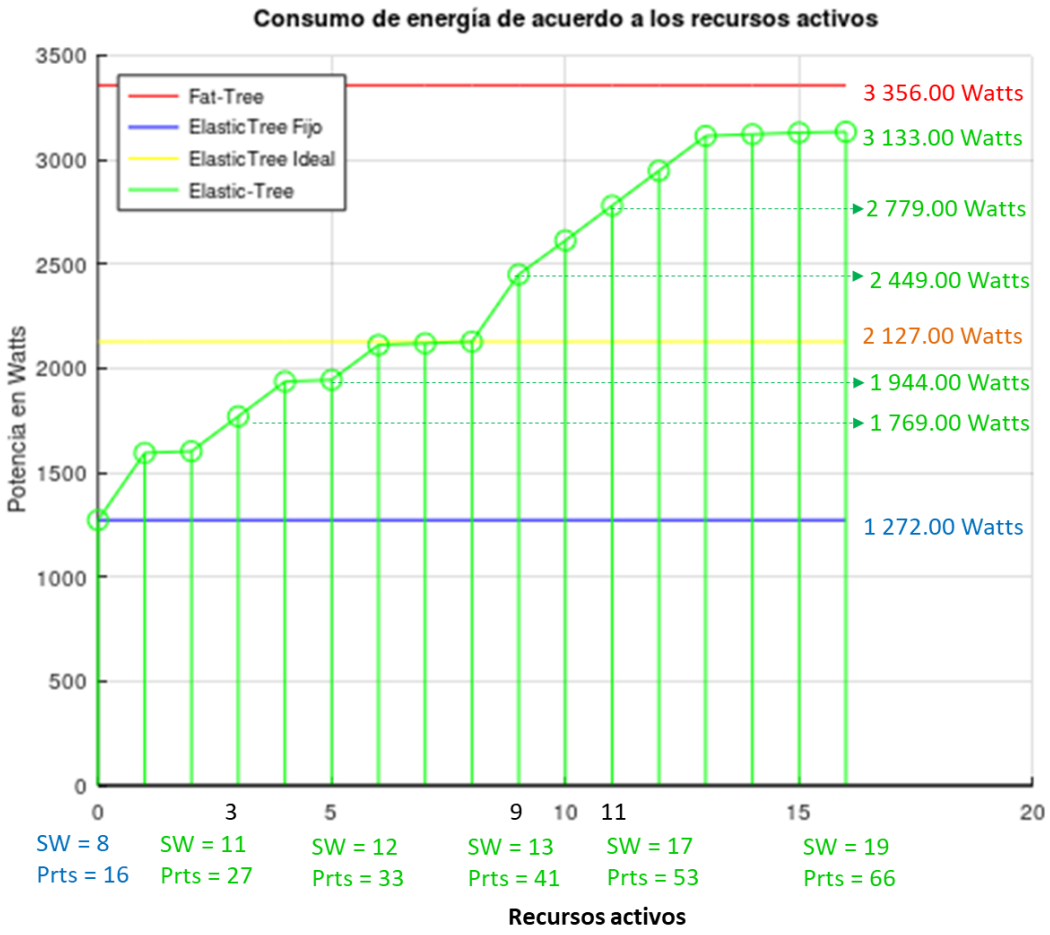


Figura 4.10: Curva de consumo de potencia vs recursos activos.

A continuación, en la Figura 4.11 podemos observar los cálculos realizados con OCTAVE utilizando las fórmulas del modelo energético para el escenario de tráfico durante una hora.

```
Escenario Trafico Durante 1 Hora

Valor medio:
-Conmutadores encendidos: 15.55
-Puertos encendidos: 48.8833
-Potencia Fija:   1272.00 Watts
-Potencia Media:   2543.58 Watts

Valor maximo:
-Conmutadores encendidos: 19
-Puertos encendidos: 66
-Potencia Fija:   1272.00 Watts
-Potencia Max:   3133.00 Watts

Ahorro de Energía:

Ahorro de Energía Elastic-Tree con trafico ideal vs Fat-Tree:
-Ahorro de Energía:     36.62 %

Ahorro de Energía Elastic-Tree con trafico vs Fat-Tree:
-Ahorro de Energía:      6.64 %

Ahorro consumo de Energía por Hora:
-Consumo de Energía Fat-Tree:   3356.00 Wh
-Consumo de Energía Elastic-Tree:   2543.58 Wh
-Ahorro en el consumo durante 1 Hora:     24.21 %

>>
```

Figura 4.11: Cálculo de la potencia consumida durante una hora.

En este caso el valor máximo de potencia alcanzada es de 3133.00 Watts, es muy superior al consumo del escenario ElasticTree Ideal de 2 172.00 Watts; pero aun así no alcanza a la potencia consumida por el escenario FatTree de 3 356.00 Watts. En este punto el ahorro de potencia del escenario ElasticTree es de 6,64 %. Si lo comparamos con el ahorro de potencia del escenario ElasticTree Ideal de 36.6%, vemos que hay una gran diferencia.

De este punto podemos concluir definitivamente que el consumo de potencia está en relación con el número de dispositivos activos, esto lo podemos ver en las Figuras 4.6 y 4.10. Por lo tanto, el ahorro de energía disminuye con el aumento de dispositivos conectados; sin embargo, en la figura 4.9 observamos que la curva de color verde varia a medida que se conectan más dispositivos en el plazo de una hora. De esta manera para

calcular el consumo de energía real, calculamos la cantidad de Watios-hora (Wh) consumidos en ese plazo.

En la figura 4.11 OTAVE nos da el resultado del cálculo de la energía consumida en Wh. Primero obtenemos el consumo de engría de FatTree durante una hora 3 356.00 Wh, y para el caso de ElasticTree el área bajo la curva de color verde es de 2 543.58 Wh. Es decir, si comparamos el funcionamiento de la red durante una hora ElasticTree consigue un ahorro del 24.21 %

4.2. Contrastación de la Hipótesis

Vamos a verificar como estos resultados validan el cumplimiento de la hipótesis planteada al inicio de este proyecto. Iniciaremos validando el cumplimento de las variables independiente y dependiente, si estas están dentro del rango estimado entonces las hipótesis especificas se han cumplido, lo que nos lleva al cumplimiento de la hipótesis general. En la tabla 4.1 observamos la hipótesis general y las específicas, así como sus indicadores que utilizaremos para la validación correspondiente.

Tabla 4.1: Tabla resumen hipótesis.

Hipótesis General		
Si aplicamos el paradigma de redes definidas por software para crear una red flexible podríamos mejorar las posibilidades de reducir el consumo de energía de los dispositivos de conectividad en la red de un Centro de Datos.		
Hipótesis Especificas		
	Es posible establecer los flujos de datos bajo demanda a través de la red de conmutadores, para controlar que puertos de comunicación estarán en uso o no durante un determinado tiempo.	
	Flexibilidad de la red para gestionar los flujos de tráfico bajo demanda, utilizando redes definidas por software.	Actualización de la tabla de flujos de los conmutadores a medida que aparecen nuevos flujos de tráfico en la red.
	Mantener los puertos de comunicación de un conmutador apagados cuando no están en uso contribuye a la reducción del consumo de energía en la red de un centro de datos.	
	Consumo de energía en la red de un centro de datos.	Reducción del consumo de energía en comparación con una topología fija como FatTree.

4.2.1. Hipótesis Especifica: Flexibilidad de la red.

Para verificar que hemos cumplido con la hipótesis específica: “Flexibilidad de la red para gestionar los flujos de tráfico bajo demanda, utilizando redes definidas por software”; verificaremos que hemos cumplido con la variable independiente.

La variable independiente es la flexibilidad de la red, que la medimos con los cambios en las tablas de flujo de los conmutadores como respuesta a una solicitud de conexión. Esto ya ha sido realizado en el capítulo 03 cuando desarrollamos el script de optimización, para validar que hemos cumplido adjuntamos las capturas de pantalla de las tablas de flujo de los conmutadores. En la figura 4.12 podemos ver como al generar tráfico hacia un host, se crea una entrada en la tabla de flujo del conmutador para poder cursarlo.

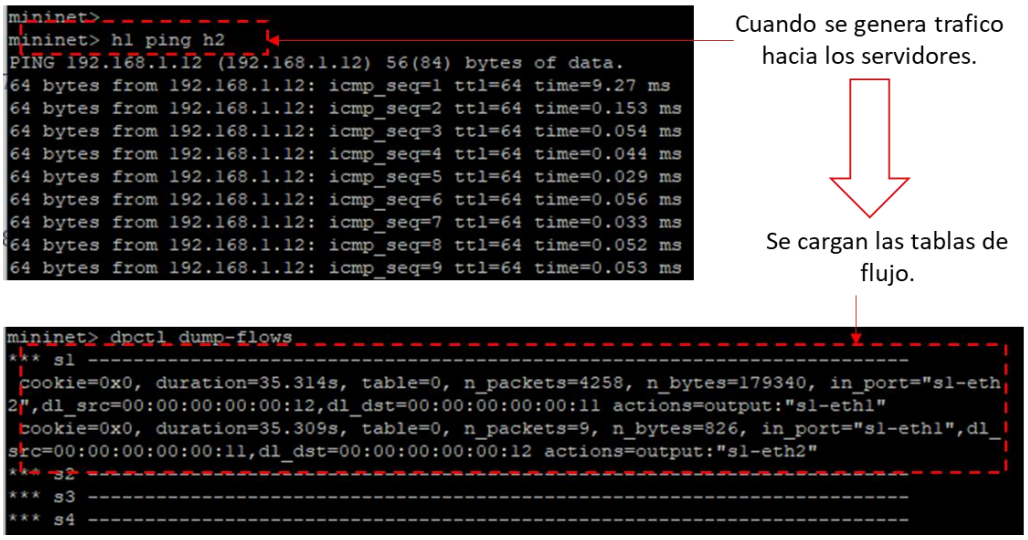


Figura 4.12: Entradas en la tabla de flujo del conmutador s1.

En conclusión, frente a un cambio en el flujo de tráfico (solicitud de conexión) modificamos la topología lógica de la red (modificación de tablas de flujo) para que se establezca un nuevo enlace y se curse el nuevo tráfico de datos. Es decir, la red es Flexible, ya que es capaz de crear flujos de tráfico bajo demanda (solicitudes de tráfico), utilizando el protocolo OpenFlow gestionado por un controlador SDN.

Con esto demostramos que se ha cumplido la primera hipótesis especifica del proyecto.

4.2.2. Hipótesis Especifica: Reducción de consumo de energía.

Para verificar que hemos cumplido con la hipótesis específica: “Mantener los puertos de comunicación de un conmutador apagados cuando no están en uso contribuye a la reducción del consumo de energía en la red de un centro de datos.”; verificaremos que hemos cumplido con la variable dependiente.

La variable dependiente es el consumo de energía del centro de datos y el indicador es la reducción del consumo de energía de este. Esta variable depende de la variable de flexibilidad medida con los enlaces en las tablas de flujo que se agregan de acuerdo con las solicitudes de tráfico. Las tablas de flujo están relacionadas con los puertos del conmutador y como vimos en el capítulo 03, estos puertos serán desactivados si no están en uso. Lo mismo se cumple con los conmutadores.

El controlador SDN es capaz de identificar y gestionar que puertos participan de la topología lógica y cuales no en un momento especifico. Entonces la topología lógica cambiará en función del tráfico que se permita cruzar en la red y solo consumirán energía los conmutadores y los puertos del conmutador que estén encendidos en ese momento. Utilizando el modelo energético y OCTAVE como herramienta de cálculo llegamos a los siguientes resultados.

```
Ahorro de Energía:

Ahorro de Energía Elastic-Tree con trafico ideal vs Fat-Tree:
-Ahorro de Energía:     36.62 %

Ahorro de Energía Elastic-Tree con trafico vs Fat-Tree:
-Ahorro de Energía:      6.64 %

Ahorro consumo de Energía por Hora:
-Consumo de Energía Fat-Tree:   3356.00 Wh
-Consumo de Energía Elastic-Tree:   2543.58 Wh
-Ahorro en el consumo durante 1 Hora:     24.21 %

>>
```

Figura 4.13: Cálculo del ahorro de energía en el centro de datos.

La red del escenario de prueba con topología FatTree n=4 funciona con 20 conmutadores y 84 puertos, consume: 3 356.00 Wh, sin importar el tráfico cursado por la red. Utilizando ElasticTree, empezamos con un escenario de 8 conmutadores y 16 enlaces

encendidos, donde no hay tráfico cursado y el consumo es de 1 272.00 Wh. Una vez que inyectamos tráfico hacia los servidores aumenta el número de recursos encendidos, hasta llegar al valor máximo de 13 conmutadores y 41 puertos encendidos. En este punto el consumo es de 2 127.00 Watts. Sin embargo, para hallar el consumo de energía durante una hora debemos calcular el área bajo la curva de color verde. OCTAVE hace estos cálculos y obtenemos: 2 543.58 Wh. En la figura 4.13 podemos ver el resultado de estos cálculos.

Como conclusión, en este escenario, podemos reducir el consumo de energía de la red del centro de datos de 3 356.00 Wh a un valor de 2 543.58 Wh, lo que nos permite un ahorro del 24.21 % en condiciones de hora pico.

Con esto demostramos que se ha cumplido la segunda hipótesis especifica del proyecto.

Para finaliza, al demostrar que se han cumplido las dos hipótesis específicas del proyecto de tesis, podemos afirmar que hemos cumplido con la hipótesis general. Es decir, utilizando redes definidas por software es posible crear una red flexible que reduzca el consumo de energía en la red de un Centro de Datos.

CONCLUSIONES

1. Es posible automatizar la actualización de los caminos que seguirán los flujos de datos bajo demanda dentro de la red del centro de datos. Esta actualización se realiza programando un controlador SDN, en nuestro caso RYU utilizando un lenguaje de programación como pyhton, que modifica las tablas de flujo de los conmutadores.

2. Es posible que el controlador ejecute algoritmos complejos como la búsqueda de rutas, en el código con python. En este caso se utilizó el algoritmo A*, para la búsqueda de rutas entre conmutadores.

3. Variando los recursos de red que permanecen encendidos, se pudo reducir el consumo de energía en un promedio del 24.21%. Dependiendo del escenario esta reducción varió entre un 6.6% hasta un 36.6%.

4. El controlador SDN Ryu, se integra con mucha facilidad a mininet y es muy configurable. Por lo que es bastante recomendable utilizarlo en entornos de prueba y entornos académicos.

5. El tiempo que le toma al puerto de un conmutador pasar del estado apagado a encendido (aprox. 35 segundos) dependiendo de la marca y modelo, puede generar una espera considerable antes que el nuevo tráfico pueda utilizar la nueva ruta establecida. Es posible configurar tiempos menores considerando las aplicaciones del centro de datos.

6. No se consideró implementar SpanningTree en el controlador, debido a que para este escenario los enlaces serán establecidos bajo demanda según el algoritmo de búsqueda y se evitarán los bucles entre conmutadores.

7. En escenarios como el uso de laboratorios remotos, en universidades, por ejemplo, esta solución permitiría encender automáticamente un grupo de equipos completo bajo demanda de acuerdo con los horarios de uso de laboratorios.

RECOMENDACIONES

1. La elección del controlador SDN debe ser en función de las necesidades específicas de la red que se desea automatizar. Ya que hay una variedad de ellos que soportan diferentes lenguajes y tienen diferentes características.

2. Aumentándole un módulo de login podría ser posible controlar el acceso a los recursos por usuario, esto permitiría aplicar este proyecto en el laboratorio remoto de una institución educativa.

3. Extender las pruebas a otras topologías de centro de datos basadas en redes CLOS, y verificar su efectividad.

4. Ampliar el proyecto utilizando diferentes algoritmos de búsqueda de rutas en función del tamaño de la red utilizada, para identificar el que tenga un mejor desempeño.

5. Realizar pruebas con equipos de hardware, de preferencia que sean White Box, para mantener la personalización y el uso de software libre.

6. Debido al tiempo que puede tomar el encendido de los puertos de un conmutador se recomienda desarrollar un algoritmo de predicción que pueda encenderlos con anticipación, adelantándose a futuros tráficos.

7. Evaluar el consumo energético en el caso del uso de más de un controlador, considerando el aumento del número de conmutadores y el uso de otros servicios como: alta disponibilidad y balanceo de carga.

8. En este proyecto se ha considerado solo tráfico ICMP y HTTP, se recomienda hacer pruebas con otros protocolos.

GLOSARIO

1. API.- Application Programming Interface, Interface para Programación de Aplicaciones.
2. CDPI.- Control-Data Plane Interface. Interface del plano de control a datos.
3. CLI.- Command Line Interface. Interface de Línea de Comandos.
4. IDE.- Integrated Development Environment. En español Entorno Integrado de Desarrollo.
5. MD-SAL.- Model-Driven Service Abstraction Layer, es la capa de abstracción de servicios de OpenDayligt donde se procesan las interacciones de los modelos que forman parte de la plataforma.
6. MV.– Máquina Virtual.
7. NBI.- NorthBound Interface. Interface de puente norte.
8. ODL.- OpenDayLight, controlador SDN de código abierto desarrollado bajo la colaboración de la Fundación Linux.
9. ONF.- Open Networking Fundation, organización que promueve el desarrollo de protocolos y componentes de código abierto para las SDN.
10. OSGi.- Open Services Gateway initiative, conjunto de estándares abiertos creado para definir las especificaciones abiertas de software para diseñar plataformas compatibles.
11. SBI.- SouthBound Interface. Interface de puente sur.
12. SDN.- Software Defined Networking, Redes Definidas por Software.
13. SO.- Sistema Operativo.
14. ToR.- Top off the Rack, en este diseño los servidores en el rack se conectan a un conmutador que está a final del rack, de tal manera que todo el cableado se queda dentro del rack.

BIBLIOGRAFIA

[1] **Al-Fares, M., Loukissas, A., Vahdat, A.** (2008). *A scalable, commodity data center network architecture.* In: Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, Seattle, pp. 63–74. ACM (2008)

[2] **Ashraf, F., Aslam, M., & Khan, Y. D.** (2018). A comprehensive comparative analysis on energy efficient approaches in SDN. Paper presented at the ACM International Conference Proceeding Series, 76-80. doi:10.1145/3206098.3206122.

[3] **Azodolmolky S.** (2013). Software Defined Networking with Openflow. Published by Packt Publishing Ltd.

[4] **Cisco. (2020).** Software-Defined Networking: Automate and program your network faster. Recuperado setiembre del 2020. https://www.cisco.com/c/en/us/solutions/software-defined-networking/overview.html

[5] **Couto R., Secci S., Mitre M., Kosmalski L.** (2015). Reliability and Survivability Analysis of Data Center Network Topologies. Springer Science+Business Media New York.

[6] **Erickson, D.** (2013). The Beacon OpenFlow Controller. Stanford University.

[7] **Gilad J.** (11 de Febrero 2020). SDN is Growing Up – It's called IBN. Cisco Blogs. [Fecha de acceso 22 de setiembre de 2020]. URL disponible en: https://blogs.cisco.com/networking/sdn-is-growing-up-its-called-ibn.

[8] **González, E.** (2015). Desarrollo de un algoritmo planificador de rutas con capacidad de implementación en diversas aplicaciones de la robótica móvil. Pereira, Colombia. [Fecha de acceso 19 de setiembre de 2020]. URL disponible en: https://core.ac.uk/download/pdf/71399082.pdf

[9] **Goransson, P., & Black, C.** (2014). Software defined networks: A comprehensive approach. San Francisco: Elsevier Science & Technology.

[10] **Halsall, F.** (Enero 2005). Computer Networking and the Internet 5th Edition. Editorial: Addison-Wesley.

[11] **Harkal V., Deshmukh A.** (2014) Software Defined Networking with Floodlight Controller. International Conference on Internet of Things, Next Generation Networks and Cloud Computing

[12] **Heller B., Seetharaman S., Mahadevan P., Yiakoumis Y, Sharma P., Banerjee S. & McKeown N.** (2010). Elastic Tree: Saving Energy in Data Center Networks. Paper presented NSDI'10 in Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation. p 17 – 17.

[13] **InternetNews.** (29 abr. 2013). Martin Casado on VMware, OpenFlow, SDN and the Future of Networking. [Fecha de acceso 10 de setiembre de 2019]. URL disponible en: Youtube. https://www.youtube.com/watch?v=Z11Uw_9f20M&t=98s

[14] **Liu Y., Muppala J., Veeraraghavan M., Lin D., Hamdi M.** (2013) Data Center Networks. Topologies, Architectures and Fault-Tolerance Characteristics. Springer.

[15] **Mahadevan P., Sharma P., Banerjee S., & Ranganathan P.** (2009). A power benchmarking framework for network devices. In Proceedings of the 8th International IFIP-TC 6 Networking Conference, NETWORKING '09, pages 795–808, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] **Mininet.org.** Mininet Overview. [Fecha de acceso 18 de setiembre de 2019]. URL disponible en: http://mininet.org/overview/

[17] **ONF: Open Networking Foundation.** (2013) SDN Architecture Overview Version 1.0 12 de Diciembre de 2013.

[18] **ONF: Open Networking Foundation.** Software-Defined Networking (SDN) Definition. [Fecha de acceso 10 de setiembre de 2019]. URL disponible en: https://opennetworking.org/sdn-definition/

[19] **ONF: Open Networking Foundation.** (2012) OpenFlow Switch Specification versión 1.3.0. 25 de Junio de 2012.

[20] **OpenDayLight - ODL.** Getting Started Guide. [Fecha de acceso 22 de enero de 2020]. URL disponible en: https://docs.opendaylight.org/en/latest/getting-started-guide/index.html

[21] **OpenDayLight - ODL.** Current Release. [Fecha de acceso 22 de enero de 2020]. URL disponible en: https://www.opendaylight.org/what-we-do/current-release/

[22] **Python Institute.** (2019). Cap: 1.1.3.1 Python - una herramienta, no un reptil. [Fecha de acceso 13 de setiembre de 2019] URL disponible en: https://edube.org/learn/programming-essentials-in-python-part-1-spanish/python-una-herramienta-no-un-reptil

[23] **Python Software Foundation.** (2019). General Python FAQ. What´s Python. [Fecha de acceso 13 de setiembre de 2019] URL disponible en: https://docs.python.org/3/faq/general.html#what-is-python

[24] **Rasmussen N.** (2012). Medición de la eficiencia eléctrica para centros de datos. White Paper 154 Rev2. Schneider Electric – Data Center Science Center.

[25] **Russell S., Pemberton D., Linton A.** (2014). RYU OpenFlow Controller. NSRC Network StartUp Resource Center. University of Oregon.

[26] **Ryu Project Team.** (2014). RYU SDN FRAMEWORK. Using OpenFlow 1.3. Release 1.0. [Fecha de acceso 15 de MARZO de 2020] URL disponible en: https://book.ryu-sdn.org/en/Ryubook.pdf

[27] **ryu.readthedocs.io**. Getting Started. (2014). [Fecha de acceso 15 de MARZO de 2020] URL disponible en: https://ryu.readthedocs.io/en/latest/getting_started.html

[28] **Sridhar R.** (2014). SDN Series Part Three: NOX, the Original OpenFlow Controller. [Fecha de acceso 22 de febrero de 2020]. URL disponible en: https://thenewstack.io/sdn-series-part-iii-nox-the-original-openflow-controller/

[29] **Yang T., Lee Y., Zomaya A.** (2014). Energy-Efficient Data Center Networks Planning with Virtual Machine Placement and Traffic Configuration. 2014 IEEE 6th International Conference on Cloud Computing Technology and Science

[30] **Zhu L., Karim MM., Sharif K., Fan Li, Du X., & Guizani M.** (2019). SDN Controllers: Benchmarking & Performance Evaluation. Computer Science. Cornell University. [Fecha de acceso 09 de marzo del 2020] URL disponible en: https://arxiv.org/abs/1902.04491

ANEXO

ANEXO 01

Scripts en pyhton, para crear la topología, el controlador y las pruebas.

- **topoFatTree4.py**

```
"""
Autor: R. Gonzales
Topologia:
FatTree n = 4
21 switch -- 16 hosts
"""
from mininet.topo import Topo

class topoFatTree4(Topo):
   def build(self):
      #Se crean 21 switches
      switch1 = self.addSwitch('s1')
       switch2 = self.addSwitch('s2')
       switch3 = self.addSwitch('s3')
       switch4 = self.addSwitch('s4')
       switch5 = self.addSwitch('s5')
       switch6 = self.addSwitch('s6')
       switch7 = self.addSwitch('s7')
       switch8 = self.addSwitch('s8')
       switch9 = self.addSwitch('s9')
       switch10 = self.addSwitch('s10')
       switch11 = self.addSwitch('s11')
       switch12 = self.addSwitch('s12')
       switch13 = self.addSwitch('s13')
       switch14 = self.addSwitch('s14')
       switch15 = self.addSwitch('s15')
       switch16 = self.addSwitch('s16')
       switch17 = self.addSwitch('s17')
       switch18 = self.addSwitch('s18')
       switch19 = self.addSwitch('s19')
       switch20 = self.addSwitch('s20')
       switch21 = self.addSwitch('s21')


   #Se crean 16 hosts
   host1 = self.addHost('h1', ip='192.168.1.11', mac='00:00:00:00:00:11')
```

```
        host2 = self.addHost('h2', ip='192.168.1.12', mac='00:00:00:00:00:12')
        host3 = self.addHost('h3', ip='192.168.1.13', mac='00:00:00:00:00:13')
        host4 = self.addHost('h4', ip='192.168.1.14', mac='00:00:00:00:00:14')
        host5 = self.addHost('h5', ip='192.168.1.15', mac='00:00:00:00:00:15')
        host6 = self.addHost('h6', ip='192.168.1.16', mac='00:00:00:00:00:16')
        host7 = self.addHost('h7', ip='192.168.1.17', mac='00:00:00:00:00:17')
        host8 = self.addHost('h8', ip='192.168.1.18', mac='00:00:00:00:00:18')
        host9 = self.addHost('h9', ip='192.168.1.19', mac='00:00:00:00:00:19')
        host10 = self.addHost('h10', ip='192.168.1.20', mac='00:00:00:00:00:20')
        host11 = self.addHost('h11', ip='192.168.1.21', mac='00:00:00:00:00:21')
        host12 = self.addHost('h12', ip='192.168.1.22', mac='00:00:00:00:00:22')
        host13 = self.addHost('h13', ip='192.168.1.23', mac='00:00:00:00:00:23')
        host14 = self.addHost('h14', ip='192.168.1.24', mac='00:00:00:00:00:24')
        host15 = self.addHost('h15', ip='192.168.1.25', mac='00:00:00:00:00:25')
        host16 = self.addHost('h16', ip='192.168.1.26', mac='00:00:00:00:00:26')

    #Host de usuarios
    host101 = self.addHost('h17', ip='192.168.1.101', mac='00:00:00:00:00:27')

        #Se crean enlaces de los servidores con los switch de acceso
        #POD01
        self.addLink(host1, switch1, 0, 1)
        self.addLink(host2, switch1, 0, 2)
        self.addLink(host3, switch2, 0, 1)
        self.addLink(host4, switch2, 0, 2)

        #POD02
        self.addLink(host5, switch3, 0, 1)
        self.addLink(host6, switch3, 0, 2)
        self.addLink(host7, switch4, 0, 1)
        self.addLink(host8, switch4, 0, 2)

        #POD03
        self.addLink(host9, switch5, 0, 1)
        self.addLink(host10, switch5, 0, 2)
        self.addLink(host11, switch6, 0, 1)
        self.addLink(host12, switch6, 0, 2)

        #POD04
        self.addLink(host13, switch7, 0, 1)
        self.addLink(host14, switch7, 0, 2)
        self.addLink(host15, switch8, 0, 1)
        self.addLink(host16, switch8, 0, 2)

        self.addLink(host101, switch21, 0, 5)

        #Se crean los enlaces por cada capa
        #Acceso a Distribucion POD01
```

```
        self.addLink(switch1, switch9, 3, 1)
        self.addLink(switch1, switch10, 4, 1)
        self.addLink(switch2, switch9, 3, 2)
        self.addLink(switch2, switch10, 4, 2)
        #Acceso a Distribucion POD02
        self.addLink(switch3, switch11, 3, 1)
        self.addLink(switch3, switch12, 4, 1)
        self.addLink(switch4, switch11, 3, 2)
        self.addLink(switch4, switch12, 4, 2)
        #Acceso a Distribucion POD03
        self.addLink(switch5, switch13, 3, 1)
        self.addLink(switch5, switch14, 4, 1)
        self.addLink(switch6, switch13, 3, 2)
        self.addLink(switch6, switch14, 4, 2)
        #Acceso a Distribucion POD04
        self.addLink(switch7, switch15, 3, 1)
        self.addLink(switch7, switch16, 4, 1)
        self.addLink(switch8, switch15, 3, 2)
        self.addLink(switch8, switch16, 4, 2)

        #Distribucion a Core POD01
        self.addLink(switch9, switch17, 3, 1)
        self.addLink(switch9, switch18, 4, 1)
        self.addLink(switch10, switch19, 3, 1)
        self.addLink(switch10, switch20, 4, 1)
        #Distribucion a Core POD02
        self.addLink(switch11, switch17, 3, 2)
        self.addLink(switch11, switch18, 4, 2)
        self.addLink(switch12, switch19, 3, 2)
        self.addLink(switch12, switch20, 4, 2)
        #Distribucion a Core POD03
        self.addLink(switch13, switch17, 3, 3)
        self.addLink(switch13, switch18, 4, 3)
        self.addLink(switch14, switch19, 3, 3)
        self.addLink(switch14, switch20, 4, 3)
        #Distribucion a Core POD04
        self.addLink(switch15, switch17, 3, 4)
        self.addLink(switch15, switch18, 4, 4)
        self.addLink(switch16, switch19, 3, 4)
        self.addLink(switch16, switch20, 4, 4)

        #Core a Monitoreo
        self.addLink(switch17, switch21, 5, 1)
        self.addLink(switch18, switch21, 5, 2)
        self.addLink(switch19, switch21, 5, 3)
        self.addLink(switch20, switch21, 5, 4)

topos = { 'topoFatTree4': ( lambda: topoFatTree4() ) }
```

- **controlRutasOptimas.py**

```
"""
Autor: R. Gonzales
Controlador RYU
"""
#Declaracion de las librerias necesarias
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ipv4
from ryu.lib.packet import in_proto

import matricesElasticTree
import algoritmoAStar

class controladorRutas(app_manager.RyuApp):

    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(controladorEnergia, self).__init__(*args, **kwargs)
        # inicializamos las tablas MAC.
        self.mac_to_port = {}

    #Esta función nos permite ingresar un nuevo flujo a los conmutadores
    def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # Construimos el flujo y lo enviamos al conmutador
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,match=match, instructions=inst)
        datapath.send_msg(mod)

    #Este evento se lanza cada vez que un conmutador se agrega al controlador
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        #Agregamos la entrada table-miss.
```

```
    #Esta entrada le permite al conmutador enviar los paquetes que no
    #coinciden con sus tablas de flujo, al controlador.
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 0, match, actions)

#Este evento se lanza cada vez que un paquete le llega al conmutador proveniente
#de un conmutador
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)

#Con esta funcion manejamos el paquete que ingresa al controlador
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # Obtenemos el Datapath ID para identificar los conmutadores.
    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    # Analizamos el paquete recibido: obtenemso la direccion origen y destino.
    pkt = packet.Packet(msg.data)
    eth_pkt = pkt.get_protocol(ethernet.ethernet)
    dst = eth_pkt.dst
    src = eth_pkt.src


    #Aqui empieza el código de la Tesis
    #1. Se verifica el destino del paquete.
    i = 0
    while i < 16:

        if matHostDir(i,0) == dst:
            nodoInicio = matHostDir(i,0)
            break
        i += 1

    #2. Se calcula el subconjunto de la matriz de trafico
    matTraficoAStarIn = getTraficoAStar(nodoInicio)

    #3. Se aplica el Algoritmo
    matNodosAStarOut = rutasAStar(matTraficoAStar)

    #4. Se carga la tabla de flujos por cada nodo en la ruta
    for nodo in matNodosAStarOut:
```

```
        actions = [parser.OFPActionOutput(out_port)]

        actualizacion=getUltimoCambio(nodo)

        #5. se instala el flujo.
        if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)

        #6. se construye el mensaje packet_out y se envía.
        out = parser.OFPPacketOut(datapath=datapath,
                                  buffer_id=ofproto.OFP_NO_BUFFER,
                                  in_port=in_port, actions=actions,
                                  data=msg.data)
        datapath.send_msg(out)

    #7. Guardamos la actualizacion en el archivo matCambiosTopo.csv
    f = open("matCambiosTopo.cvs", "a")
    f.write(actualizacion)
    f.close()
```

- **testElasticTree.py**

```
"""
Autor: R. Gonzales
Prueba ElaticTree:
60 flujos de tráfico

"""

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.util import dumpNodeConnections
from mininet.log import setLogLevel
import time
import topoFatTree4

#Esta funcion carga la topologia, inicia mininet
#inicia la red, ejecuta el CLI, detiene la red
def testElasticTree():
   #Creamos la matriz de prueba:
   matTestIP["h1","h1","h1","h3","h8","h8","h10","h10","h10","h10","h4","h2","h15","h2","h2","h12","h12","h5","h
9","h9","h5"]

   #Crear la topologia FatTree n=4
   topo = topoFatTree4()
   controller=remote
```

```
    net = Mininet(topo,controller)

    #Se inicia la red
    print ("Iniciando la red")
    net.start()
    time.sleep(2)

    #Creamos un ping hacia cada direccion de la matriz de prueba
    print ("Iniciando Pruebas de ping")
    hostOrig = net.get('h27')
    hostOrig_IP = hostOrig.IP()
    for ipd in matTestIP:
        hostDest = net.get(idp)
        hostDest_IP = hostDest.IP()
        testCmd = 'ping -c10 %s' % hostDest_IP
        hostOrig.cmd(testCmd)
        print("Ping a: ", hostDest_IP)
        time.sleep(10)

    #Se detiene la red
    print ("Deteniendo la red")
    net.stop()

if __name__ == '__main__':
    setLogLevel('info')
    testElasticTree()
```

- **algoritmoAStar.py**

```
"""
Autor: R. Gonzales
A*
"""

#Funcion que calcula la ruta final
def rutasAStar(listaSwIn, matTrafico):

    #Función que presenta el valor de Fn de la lista de nodos de la lista abierta.
    def valor_fn(lista):
        nodo = lista
        return nodo[2]

    #Función heuristica que calcula el valor de hn
    def funHn(nodo, mat):
        hni = []
        indiceFin = len(mat[nodo]) - 1
        for i in range(nodo+1,len(mat[nodo])):
            if mat[nodo][i] > 0:
                hni.append(mat[nodo][i] + mat[i][indiceFin])

        hn = min(hni, default = 0)
        return hn

    #Funcion para convertir nodos a conmutaodres de la ruta escogida
    def converNodo2Sw(listaNodos,listaSwTotal):
        listaSw = []
        for i in listaNodos:
            listaSw.append(listaSwTotal[i])
```

```
    return listaSw

#Función que calcula los nodos de la lista abierta en funcion del nodo actual.
def listaAbiertaCrear(nodoActual, nodoFin, matNodos):
    listaAbierta = []
    hn = 0
    gn = 0

    for n in range(nodoActual+1,len(matNodos)):
        if matNodos[nodoActual][n] > 1:
            if n != nodoFin:
                hn = funHn(n, matNodos)
            gn = matNodos[nodoActual][n]
            fn = hn + gn
            listaAbierta.append([n, nodoActual, fn, gn, hn])

    return listaAbierta

listaAbierta = []
listaCerrada = []
rutaAStar = []
listaRutaNodos = []
listaRutaSw = []
nodoActualId = 0
nodoFinId = len(listaSwIn)-1
nodoActualPadre = nodoActualId
nodoActual = [nodoActualId,nodoActualPadre]

listaCerrada.append(nodoActual)
listaAbierta = listaAbiertaCrear(nodoActualId,nodoFinId,matTrafico)

while len(listaAbierta) > 0:
    listaNodoActual = min(listaAbierta, key=valor_fn)
    #print("Lista Nodo Elegido:",listaNodoActual)
    nodoActualId = listaNodoActual[0]
    nodoActualPadre = listaNodoActual[1]
    nodoActual = [nodoActualId,nodoActualPadre]
    listaCerrada.append(nodoActual)
    listaAbierta = listaAbiertaCrear(nodoActualId,nodoFinId,matTrafico)


#Calculo de la ruta a partir de la lista cerrada
listaCerradaRev = list(reversed(listaCerrada))

parentId = nodoFinId
for n in range(0,len(listaCerradaRev)):
    if listaCerradaRev[n][0] == parentId:
        listaRutaNodos.append(listaCerradaRev[n][0])
    parentId = listaCerradaRev[n][1]

listaRutaSw = converNodo2Sw(listaRutaNodos,listaSwIn)

return listaRutaSw
```

ANEXO 02

Scripts en OCTAVE, para el cálculo energético.

- **calculoEnergéticoBase.m**

```
#Calculo Energético 1.0
#Escenarios Basicos

#Limpiamos memoria, cerramos archivos y limpiamos CLI
clear all
close all
clc

#Archivo que contiene los datos
nomArchivo = 'matCambiosTopo.xlsx';
nomHoja = 'activos_01';

#Valores energéticos predefinidos en Watts
Pch = 151;
Pi = 3.83;
Pa = 4;

#Valores de la topologia Fat-Tree
n=4;
numSWTotal = (5/4)*n^2;
numSWAcceso = (n^2)/2;
numSWDistro = (n^2)/2;
numSWCore = (n^2)/4;
numServidores = (n^3)/4;
numPuertosTotal = (5/4)*n^3 + numSWCore;

disp('Inicamos el calculo energético...');
#Escenario Fat-Tree
PotFatTree = numSWTotal*Pch + numPuertosTotal*Pa;

#Calculamos la potencia consumida por los dispositivos siempre encendidos
#Para hallar el consumo fijo de potencia
#Conmutadores acceso:
numSWFijo = numSWAcceso;
numPuertosFijo = numServidores;

PotFija = numSWFijo*Pch + numPuertosFijo*Pa;

#Cargamos la matriz con el dato de puertos y conmutadores activos
pkg load io;
matActivos = xlsread(nomArchivo,nomHoja);

#Procesamos la matriz de de entrada
vecActivos = sum(matActivos);

#Escenario Elastic-Tree sin trafico
numSWOn01 = vecActivos(1);
numPuertosOn01 = vecActivos(2);

PotVariable01 = numSWOn01*Pch + numPuertosOn01*Pa;
PotTotal01 = PotFija + numSWOn01*Pch + numPuertosOn01*Pa;

#Escenario Elastic-Tree con trafico
numSWOn02 = vecActivos(3);
numPuertosOn02 = vecActivos(4);

PotVariable02 = numSWOn02*Pch + numPuertosOn02*Pa;
PotTotal02 = PotFija + numSWOn02*Pch + numPuertosOn02*Pa;
```

```
#Eficiencia energética Elastic-Tree vs Fat-Tree
ahorroEnergia = 1 - PotTotal02/PotFatTree;


disp('');
disp('Escenarios Básicos');
disp('');
disp('Escenario 01: Fat-Tree');
printf('-Conmutadores encendidos: %d\n', numSWTotal);
printf('-Puertos encendidos: %d\n', numPuertosTotal);
printf('-Potencia Total: %9.2f Watts\n', PotFatTree);
disp('');
disp('Escenario 02: Elastic-Tree sin trafico');
printf('-Conmutadores encendidos: %d\n', numSWFijo);
printf('-Puertos encendidos: %d\n', numPuertosFijo);
printf('-Potencia Fija: %9.2f Watts\n', PotFija);
printf('-Potencia Variable: %9.2f Watts\n', PotVariable01);
printf('-Potencia Total: %9.2f Watts\n', PotTotal01);
disp('');
disp('Escenario 03: Elastic-Tree con trafico ideal');
printf('-Conmutadores encendidos: %d\n', numSWFijo+numSWOn02);
printf('-Puertos encendidos: %d\n', numPuertosFijo+numPuertosOn02);
printf('-Potencia Fija: %9.2f Watts\n', PotFija);
printf('-Potencia Variable: %9.2f Watts\n', PotVariable02);
printf('-Potencia Total: %9.2f Watts\n', PotTotal02);
disp('');
disp('Ahorro de energía Elastic-Tree con trafico ideal vs Fat-Tree');
printf('-Ahorro de energía: %9.2f %%\n', ahorroEnergia*100);
disp('');
```

- **calculoEnergéticoElasticTree.m**

```
#Calculo Energético 1.0
#Tráfico Elastic Tree Trafico Hora Pico

#Limpiamos memoria, cerramos archivos y limpiamos CLI
clear all
close all
clc

#Archivo que contiene los datos
nomArchivo = 'matCambiosTopo.xlsx';
nomHojaIni = 'activos_01';
nomHojaFlujo = 'activos_03';

#Valores energéticos predefinidos
Pch = 151;
Pi = 3.83;
Pa = 4;

#Valores de la topologia Fat-Tree
n=4;
numSWTotal = (5/4)*n^2;
numSWAcceso = (n^2)/2;
numSWDistro = (n^2)/2;
numSWCore = (n^2)/4;
numServidores = (n^3)/4;
numPuertosTotal = (5/4)*n^3 + numSWCore;

disp('Inicamos el calculo energético...');
#Escenario Fat-Tree
PotFatTree = numSWTotal*Pch + numPuertosTotal*Pa;

#Calculamos la potencia consumida por los dispositivos siempre encendidos
```

```
#Para hallar el consumo fijo de potencia
#Conmutadores acceso:
numSWFijo = numSWAcceso;
numPuertosFijo = numServidores;

PotFija = numSWFijo*Pch + numPuertosFijo*Pa;

#Cargamos la matriz con el dato de puertos y conmutadores activos
pkg load io;
matActivosIni = xlsread(nomArchivo,nomHojaIni);
matActivosFlujo = xlsread(nomArchivo,nomHojaFlujo);

#Procesamos la matriz de de entrada
vecActivosIni = sum(matActivosIni);
vecActivosFlujo = sum(matActivosFlujo);

#Escenario Elastic-Tree sin trafico
numSWOn01 = vecActivosIni(1);
numPuertosOn01 = vecActivosIni(2);

PotVariable01 = numSWOn01*Pch + numPuertosOn01*Pa;
PotTotal01 = PotFija + numSWOn01*Pch + numPuertosOn01*Pa;

#Escenario Elastic-Tree con trafico
numSWOn02 = vecActivosIni(3);
numPuertosOn02 = vecActivosIni(4);

PotVariable02 = numSWOn02*Pch + numPuertosOn02*Pa;
PotTotal02 = PotFija + numSWOn02*Pch + numPuertosOn02*Pa;

#Preparamos los puntos para graficar la potencia
k=0;
#Puntos de tiempo
t=0:length(vecActivosFlujo)/2-1;
#Puntos del flujo de potencia de Fat-Tree
PotTotalFatTree=PotFatTree*t.^0;
#Puntos del flujo de potencia de Elastic-Tree Sin Trafico
PotFijaElasticTree=PotTotal01*t.^0;
#Puntos del flujo de potencia de Elastic-Tree Ideal
PotElasticTreeIdeal=PotTotal02*t.^0;

for i=0:length(vecActivosFlujo)/2-1
  #Puntos del flujo de potencia
  numSWOnVar = vecActivosFlujo(i+1+k);
  numPuertosOnVar = vecActivosFlujo(i+2+k);
  PotTotalFunc(1,i+1) = PotFija + numSWOnVar*Pch + numPuertosOnVar*Pa;
  PotTotalFunc(2,i+1) = numSWOnVar;
  PotTotalFunc(3,i+1) = numPuertosOnVar;
  k=i+1;
endfor
PotTotalFunc
#Hallamos la potencia maxima alcanzada
[Pmax,tmax]=max(PotTotalFunc(1,:));
Pm=mean(PotTotalFunc(1,:));
mSW=mean(PotTotalFunc(2,:));
mPuertos=mean(PotTotalFunc(3,:));

#Hallamos el aumento de potencia por equipos activos
PotTotalRecurso=unique(PotTotalFunc(1,:))
p=0:length(PotTotalRecurso)-1;
PotTotalFatTree2=PotFatTree*p.^0;
PotFijaElasticTree2=PotTotal01*p.^0;
PotElasticTreeIdeal2=PotTotal02*p.^0;

#Eficiencia energética Elastic-Tree vs Fat-Tree
ahorroEnergiaIdeal = 1 - PotTotal02/PotFatTree;
```

```
ahorroEnergiaFin = 1 - Pmax/PotFatTree;

consumoEnergiaxHoraFT = PotFatTree;
consumoEnergiaxHoraET = sum(PotTotalFunc(1,:))/60;

ahorroConsumoxHora = 1 - consumoEnergiaxHoraET/consumoEnergiaxHoraFT;

#Creamos las gráficas
#Figura 1: Consumo por una hora
figure;
grid on;

xlabel("Tiempo en minutos");
ylabel("Potencia en Watts");
title("Consumo de energía durante 1 hora");

hold on;
#Limites superior e inferior
plot(t,PotTotalFatTree,'r');
plot(t,PotFijaElasticTree,'b');
plot(t,PotElasticTreeIdeal,'y');

#Flujo de potencia
plot(t,PotTotalFunc(1,:),'g');
legend('Fat-Tree','ElasticTree Fijo','ElasticTree Ideal','Elastic-Tree',"Location","northwest");
hold off;

figure;
grid on;

xlabel("Recursos activos");
ylabel("Potencia en Watts");
title("Consumo de energía de acuerdo a los recursos activos");
hold on;
#Limites superior e inferior
plot(p,PotTotalFatTree2,'r');
plot(p,PotFijaElasticTree2,'b');
plot(p,PotElasticTreeIdeal2,'y');
plot(p,PotTotalRecurso,'g');
stem(p,PotTotalRecurso,'g');
legend('Fat-Tree','ElasticTree Fijo','ElasticTree Ideal','Elastic-Tree',"Location","northwest");
hold off;

disp('');
disp('Escenarios Basicos');
disp('');
disp('Escenario 01: Fat-Tree');
printf('-Conmutadores encendidos: %d\n', numSWTotal);
printf('-Puertos encendidos: %d\n', numPuertosTotal);
printf('-Potencia Total: %9.2f Watts\n', PotFatTree);
disp('');
disp('Escenario 02: Elastic-Tree sin trafico');
printf('-Conmutadores encendidos: %d\n', numSWFijo);
printf('-Puertos encendidos: %d\n', numPuertosFijo);
printf('-Potencia Fija: %9.2f Watts\n', PotFija);
printf('-Potencia Variable: %9.2f Watts\n', PotVariable01);
printf('-Potencia Total: %9.2f Watts\n', PotTotal01);
disp('');
disp('Escenario 03: Elastic-Tree con trafico ideal');
printf('-Conmutadores encendidos: %d\n', numSWFijo+numSWOn02);
printf('-Puertos encendidos: %d\n', numPuertosFijo+numPuertosOn02);
printf('-Potencia Fija: %9.2f Watts\n', PotFija);
printf('-Potencia Variable: %9.2f Watts\n', PotVariable02);
printf('-Potencia Total: %9.2f Watts\n', PotTotal02);
disp('');
disp('');
```

```
disp('Escenario Trafico Durante 1 Hora');
disp('');
disp('Valor medio:');
printf('-Conmutadores encendidos: %d\n', numSWFijo+mSW);
printf('-Puertos encendidos: %d\n', numPuertosFijo+mPuertos);
printf('-Potencia Fija: %9.2f Watts\n', PotFija);
printf('-Potencia Media: %9.2f Watts\n', Pm);
disp('');
disp('Valor maximo:');
printf('-Conmutadores encendidos: %d\n', numSWFijo+PotTotalFunc(2,tmax));
printf('-Puertos encendidos: %d\n', numPuertosFijo+PotTotalFunc(3,tmax));
printf('-Potencia Fija: %9.2f Watts\n', PotFija);
printf('-Potencia Max: %9.2f Watts\n', Pmax);
disp('');
disp('Ahorro de Energía:');
disp('');
disp('Ahorro de Energía Elastic-Tree con tráfico ideal vs Fat-Tree:');
printf('-Ahorro de Energía: %9.2f %%\n', ahorroEnergiaIdeal*100);
disp('');
disp('Ahorro de Energía Elastic-Tree con tráfico vs Fat-Tree:');
printf('-Ahorro de Energía: %9.2f %%\n', ahorroEnergiaFin*100);
disp('');
disp('Ahorro consumo de Energía por Hora:');
printf('-Consumo de Energía Fat-Tree: %9.2f Wh\n', consumoEnergiaxHoraFT);
printf('-Consumo de Energía Elastic-Tree: %9.2f Wh\n', consumoEnergiaxHoraET);
printf('-Ahorro en el consumo durante 1 Hora: %9.2f %%\n', ahorroConsumoxHora*100);
disp('');
```